UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Primož Remic

Optimizacija komunikacije med krmilnikom in stikali omrežja SDN z uporabo razpršenega oddajanja krmilnih sporočil

MAGISTRSKO DELO

MENTOR: prof. dr. Nikolaj Zimic

Ljubljana, 2016

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Večna pot 113
1000 Ljubljana, Slovenija
telefon: 01 47 98 100
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Številka: 161-MAG-RI/2016
Datum: 06. 04. 2016

Primož REMIC, univ. dipl. inž. rač. in inf.

L j u b l j a n a

Fakulteta za računalništvo in informatiko Univerze v Ljubljani izdaja naslednjo magistrsko nalogo

Naslov naloge: **Optimizacija komunikacije med krmilnikom in stikali omrežja SDN z uporabo razpršenega oddajanja krmilnih sporočil**

Optimization of communication between controller and switches of SDN network by multicasting controller messages

Tematika naloge:

V zadnjem času na področju informacijskih omrežij postaja vse bolj prodorna tehnologija programsko definiranih omrežij (ang. "Software Defined Network", SDN). Tehnologija prinaša izvzetje kontrolne ravnine iz vseh omrežnih naprav v določenem delu omrežja. Kontrolna ravnina se kot enotna nadzorna aplikacija izvaja na oddaljenem strežniku. Na omrežnih elementih ostaja samo še podatkovna ravnina, zadolžena izključno za posredovanje podatkovnih paketov. Za komunikacijo med kontrolno ravnino in množico podatkovnih ravnin se uporablja različne protokole. Preko teh protokolov kontrolna ravnina nadzira podatkovne ravnine in jim pošilja množico me seboj podobnih krmilnih sporočil. Način komunikacije je možno optimizirati z uporabo sporočil drugačnega formata, ki bodo vsebovali enotni ukaz za vse nadzirane naprave in jih bo možno pošiljati razpršeno.

V magistrski nalogi izdelajte protokol, ki bo omogočal optimalnejšo komunikacijo med krmilnikom in stikali SDN omrežja z uporabo razpršenega oddajanja enotnih krmilnih sporočil. V nalogi najprej raziščite delovanje SDN omrežij in možnosti simulacije le-tega. Nato preučite koncepte delovanja ključnih protokolov znotraj omrežja in opredelite možnosti za njihovo nadgradnjo in optimalnejše izvajanje. V simuliranem okolju izdelajte in preizkusite izboljšano rešitev. Ustreznost rešitve ovrednotite na podlagi primerjave opravljenih meritev na originalnem in izboljšanem protokolu.

M e n t o r :

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

D e k a n :

prof. dr. Nikolaj Zimic

Zahvaljujem se mentorju prof. dr. Nikolaju Zimicu za strokovno vodenje in spodbudo pri nastajanju magistrskega dela.

Hvala podjetju Iskratel, d. o. o., za vsa pridobljena znanja in izkušnje, ki so mi pri mojem delu zelo koristila.

Zahvala gre tudi vsem domačim in sodelavcem, ki so z nasveti, besedami spodbude ali kakorkoli drugače pripomogli k nastajanju tega dela.

Za podporo in razumevanje ob nastajanju dela se zahvaljujem ženi Katji ter sinovoma Roku in Urhu, ki sta ves čas pogumno prenašala dejstvo, da jima namenjam bistveno manj časa kot običajno.

Hvala tudi ge. Sabini Leben za hitro in strokovno lektoriranje dela ter ge. Darji Pretnar za pregled delov, napisanih v angleškem jeziku.

Posvečeno moji družini.

KAZALO

Okrajšave in ključne besede:

ACL	*Access Control List*
API	*Application programming interface*
ARP	*Address Resolution Protocol*
BGP	*Border Gateway Protocol*
CAM	*Context Addressed Memory*
CLI	*Command Line Interface*
FCS	*Frame Check Sequence*
ForCES	*FORwarding & Control Element Separation*
FPGA	*Field-programmable Gate Array*
HAL	*OpenFlow Hardware Abstraction Layer*
HTTP	*HyperText Transfer Protocol*
L2	*OSI Level 2 – Link Layer*
L3	*OSI Level 3 – Network Layer*
L4	*OSI Level 4 – Transport Layer*
LAN	*Local Area Network*
IEEE	*Institute of Electrical and Electronics Engineers*
IP	*Internet Protocol*
IPv6	*Internet Protocol version 6*
OF	*OpenFlow protocol*
ONF	*Open Networking Foundation*
OSI	*Open Systems Interconnection*
OVS	*Open vSwitch*
POF	*Protocol Oblivious Forwarding*
PoC	*Proof of Concept*
RAM	*Random-access Memory*
RMT	*Reconfigurable Match Tables*
MAC	*Media Access Control*
MPLS	*Multi-Protocol Label Switching*
NFV	*Network Function Virtualization*
SDN	*Software Defined Networks*
SSH	*Secure Shell*
TCP	*Transmission Control Protocol*
ToS	*Type of Service*
UDP	*User Datagram Protocol*
VLAN	*Virtual LAN*

Povzetek

Velika večina današnjih računalniških omrežij je visoko decentraliziranih. Vsak od elementov omrežja deluje popolnoma avtonomno s pomočjo lastne krmilne programske opreme. Elementi medsebojno sporočajo in zaznavajo stanja z uporabo množice kompleksnih protokolov, ki omogočajo, da takšno omrežje pravilno deluje. Edina centralizirana točka teh omrežij je upravljanje konfiguracijskih podatkov, vendar še ta ni nujno potrebna, saj vsak od elementov omogoča tudi avtonomno upravljanje.

Zadnjih nekaj let pospešeno nastaja popolnoma drugačna zasnova računalniških omrežij. Omrežja prihodnosti naj bi bila v največji možni meri centralizirana. Centraliziran krmilnik takšnega omrežja v vsakem trenutku opredeli delovanje vsakega od elementov v omrežju v skladu s trenutnim programom, ki ga izvaja. Na ta način je preko ene točke možno programsko opredeliti delovanje celotnega omrežja, zato se ta omrežja imenujejo: »Programsko opredeljena omrežja« (SDN).

V magistrskem delu je podrobno opisana arhitektura in princip delovanja programsko opredeljenih omrežij. Utemeljene so prednosti in slabosti takšnih omrežij. Osrednji del dela je predstavitev predloga za novo in optimalnejšo metodo krmiljenja omrežij SDN. Metoda prinaša občutne izboljšave nekaterih pomembnih slabosti programsko opredeljenih omrežij, kot so slaba zmogljivost in razširljivost krmilnika ter pomanjkljivo izpolnjevanje zahtev po delovanju v realnem času.

Predlagana metoda obsega definicijo univerzalnega krmilnega sporočila, ki za namen vzpostavitve nove povezave lahko krmili več elementov omrežja. Dopolnjena izvedba krmilnika omogoča, da takšno sporočilo odda tako, da ta doseže vse vpletene elemente oziroma stikala SDN. Spremenjena zasnova stikal omogoča, da takšno sporočilo ustrezno interpretirajo.

Predstavljeno je hipotetično omrežje SDN, ki je uporabljeno kot osnova za teoretični izračun izboljšane učinkovitosti nove metode v primerjavi z obstoječimi. Izračun dokazuje veliko izboljšanje prej omenjenih pomanjkljivosti programsko opredeljenih omrežij.

Postavljeno je bilo tudi simulacijsko okolje, v katerem smo preizkusili delovanje nove metode. Opravljene simulacije so v primeru krmiljenja po novi metodi pokazale izboljšanje večine značilnosti omrežja. Pri nekaterih značilnostih so se močneje izrazila odstopanja postavljenega simulacijskega okolja od stanja v realnih omrežjih, zato s pomočjo simulacij ni bilo možno izkazati vseh prednosti predlagane metode krmiljenja.

Abstract

Most of today's computer networks are highly decentralized. Each network element operates completely autonomously using its own control software. Network elements communicate with each other and detect each other's state using a variety of complex protocols that enable the network to operate properly. The only centralized point of such networks is the management plane, although it is not essential either, since autonomous management is provided by all network elements.

In recent years we have witnessed a rapid emergence of a totally different network concept. The network of the future is highly centralized. The centralized controller's software of such network defines the behaviour of every network element at any time. In this way the whole network is defined by the software, through a single control point. The concept is known as »Software Defined Networks« (SDN).

In the master thesis we describe the architecture and principle of operations of software defined networks in detail. We substantiate the advantages and disadvantages of such networks. The main part of the thesis is a proposal of a new, more optimal, method for controlling SDN. The method brings significant improvements of certain important weaknesses of software defined networks, such as poor performance and scalability of the controller and the lack of compliance with real-time requirements.

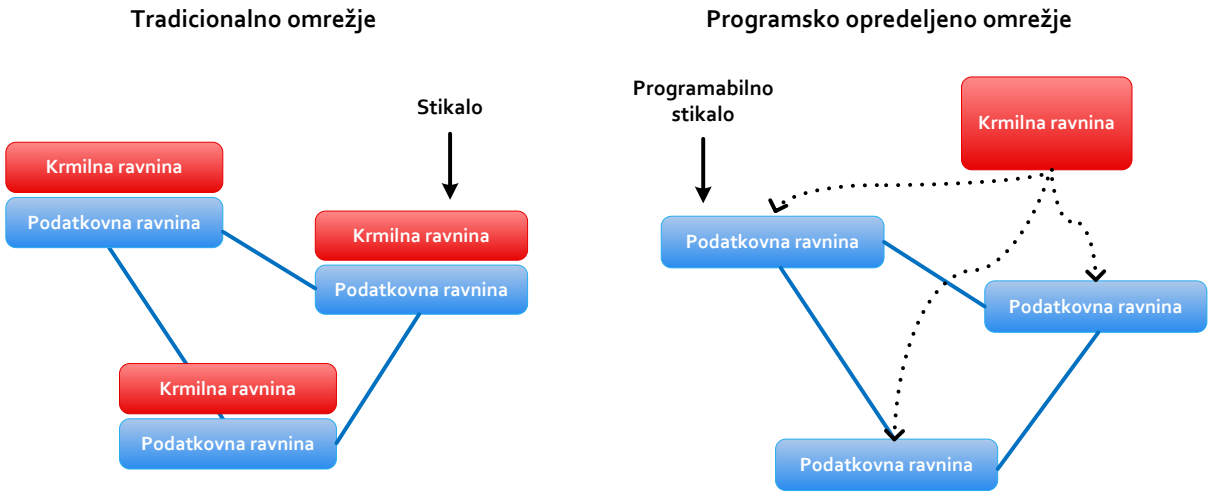
The proposed method comprises a universal definition of the control message, which enables the establishment of a new flow forwarding path through multiple network elements. The modification of the controller makes it possible to cast such a message so that it reaches all relevant elements or SDN switches. Moreover, adjustable design of the switches further contributes to proper interpretation of these messages.

Additionally, we present a hypothetical network which was used as a basis for theoretical calculations of efficiency improvement of the new method, compared to the existing ones. The calculations show a substantial improvement of the aforementioned drawbacks of software defined networks.

Finally, we have put up a simulation environment for testing of the new method. The performed simulations have showed improvement in most network characteristics. Deviations between the simulated and real network environments have come into consideration with the other characteristics. Consequently, by running the simulation it was not possible to demonstrate all the advantages of the proposed control method.

1 UVOD

Programsko opredeljena omrežja – SDN (ang. *Software Defined Networks*) so v zadnjih letih deležna velike pozornosti, ker prinašajo možnost programiranja računalniških omrežij ter s tem lažje in hitrejše uvajanje novih inovativnih rešitev. SDN uvaja ločitev podatkovne in kontrolne oz. krmilne ravnine (Slika 1), in tako olajša implementacijo kompleksnih omrežnih funkcij. Pričakujemo, da bo strojna oprema bolj splošno namenska in cenejša ter da jo bo možno krmiliti preko standardnih vmesnikov. Predvidevamo tudi večjo fleksibilnost, tako da bo možno dinamično dodajati nove omrežne funkcije v obliki omrežnih aplikacij, podobno kot poteka nalaganje aplikacij na mobilne telefone. To bi omogočilo modularizacijo in neodvisen razvoj omrežnih funkcij.



Slika 1: Ločitev podatkovne in krmilne ravnine

Ločitev krmilne in podatkovne ravnine pomeni, da se krmilna ravnina kot centralizirana nadzorna aplikacija izvaja na oddaljenem strežniku. Centralizirana je za vse usmerjevalnike in stikala v izbranem omrežju. V teh napravah tako ni več logike za izvajanje omrežnih protokolov, temveč ostaja samo še podatkovna ravnina, katere glavna naloga je posredovanje podatkovnih paketov. Za komunikacijo med obema ravninama se uporablja različne standardne in nestandardne ter odprte in zaprte protokole. Želimo si, da so ti protokoli čim bolj standardni in čim bolj odprti, zato na tem področju dominira protokol *OpenFlow* (poglavje 2.6).

V preteklem obdobju so bile v literaturi izpostavljene predvsem prednosti omrežja SDN, kot so, že omenjena fleksibilnost, enostavnost, neodvisnost, večja odprtost, ipd. (poglavje 2.7). V zadnjem času so praktične implementacije pokazale tudi na slabosti omrežja SDN (poglavje 2.8). Kot glavna slabost se omenja slaba razširljivost oz. skalabilnost, ki je posledica velike obremenitve krmilne ravnine z rastjo omrežja. Naraščajoča obremenitev krmilne ravnine je neposredno povezana s povečevanjem omrežja, saj tako narašča potreba po interakciji z vedno večjim številom podatkovnih ravnin.

V magistrskem delu je podan predlog za spremembo metode komunikacije med obema ravninama tako, da se zmanjša potrebna interakcija med njima. Posledično se zmanjša obremenitev krmilne ravnine in zakasnitev pri posredovanju prvih paketov posameznega podatkovnega toka na podatkovni ravnini. Cilj je, da se nov predlog ustrezno teoretično razdela, praktično implementira in preizkusi v simulacijskem okolju ter ustrezno opredeli rezultate preizkusa.

Naslednje poglavje podaja osnovna definicija omrežja SDN in razlage povezanih pojmov. Podane so tudi prednosti oz. motivi za uporabo SDN, kot tudi omejitve, ki izhajajo iz prakse. Sledi definicija protokola *OpenFlow*, omenjene pa so tudi možne alternative tega protokola.

V tretjem poglavju sta predstavljeni dve obstoječi metodi komunikacije oz. metodi krmiljenja med obema ravninama. Četrto poglavje zajema podrobno predstavitev nove in optimalnejše metode krmiljenja. Opisan je princip delovanja nove metode, spremembe, ki so potrebne na obeh ravninah za implementacijo metode ter njene dobre in slabe strani. Predstavljen je tudi izračun pričakovanih izboljšav za posamezne značilnosti omrežja ter izračun teoretične učinkovitosti nove metode na hipotetičnem omrežju.

Peto poglavje obsega predstavitev izbranega okolja za preizkus predlagane metode v simulacijskem okolju in razlago kode, razvite v ta namen. V šestem poglavju sta predstavljena potek simulacije ter predstavitev in interpretacija njenih rezultatov. Dodatek vsebuje še celotno programsko kodo, razvito v okviru magistrskega dela.

2 PROGRAMSKO OPREDELJENA OMREŽJA – SDN

V zadnjih dvajsetih letih je bilo razvitih že vrsto sistemov, ki so vsak na svoj način realizirali idejo programsko opredeljenih omrežij, ali vsaj njen del [9]. Izraz *Software Defined Networks* je bil prvič uporabljen leta 2009 v članku [29], ki govori o projektu na univerzah Stanford in Berkeley, imenovanem *OpenFlow*. *OpenFlow* (v nadaljevanju tudi OF) je dominantni protokol za komunikacijo med podatkovno in krmilno ravnino v omrežjih SDN, ki se v novejšem času vzdržuje pod okriljem organizacije Open Networking Funation (ONF).

Vse od predstavitve protokola OF [17] zanimanje za tehnologije, povezane s programsko opredeljenimi omrežji, strmo narašča. Tako sta SDN in sestrska tehnologija navidezne omrežne funkcije – NFV (ang. *Network Function Virtualization*) zadnja leta najpomembnejši temi v strokovnih krogih. Takšna popularnost tehnologije ima logično posledico, da različne interesne skupine, predvsem veliki proizvajalci opreme in veliki operaterji, skušajo s svojimi predlogi vplivati tudi na preoblikovanje zasnove tehnologije. Tako obstajajo tudi alternativne definicije omrežja SDN, ki predlagajo drugačne metode, s pomočjo katerih se uresniči vsaj nekaj ciljev SDN. Dva primera takšne definicij sta SDN preko obstoječih vmesnikov (ang. SDN *via Existing* APIs) in SDN preko prekrivnih omrežij (ang. SDN *via Overlay Networks*) [11].

Ciljem SDN v celoti sledi osnovna definicija, ki izhaja iz predlogov, podanih ob predstavitvi protokola OF, in jo predpisuje organizacija ONF [32]. Magistrsko delo se v celoti opira na to definicijo, ki je tudi najbolj široko priznana – podpira jo celotna akademska sfera in tudi velik del industrije.

2.1 Definicija

Programsko opredeljena omrežja so nov pristop pri gradnji in upravljanju omrežij. Omogoča implementacijo novih omrežnih storitev s programiranjem nizko nivojskih posredovalnih funkcij. To je doseženo s fizično razdružitvijo sistema, ki odloča, kam se posamezen promet posreduje – krmilna ravnina, ter sistema, ki dejansko izvede posredovanje prometa v skladu z odločitvijo krmilne ravnine – podatkovna ali posredovalna ravnina. Krmilna ravnina je nadalje logično združena v (logično) enovit krmilnik, ki krmili potek prometa v celotnem omrežju SDN. Takšna arhitektura omogoča bolj dinamična, laže nadgradljiva, za upravljanje enostavnejša in cenovno učinkovitejša omrežja.

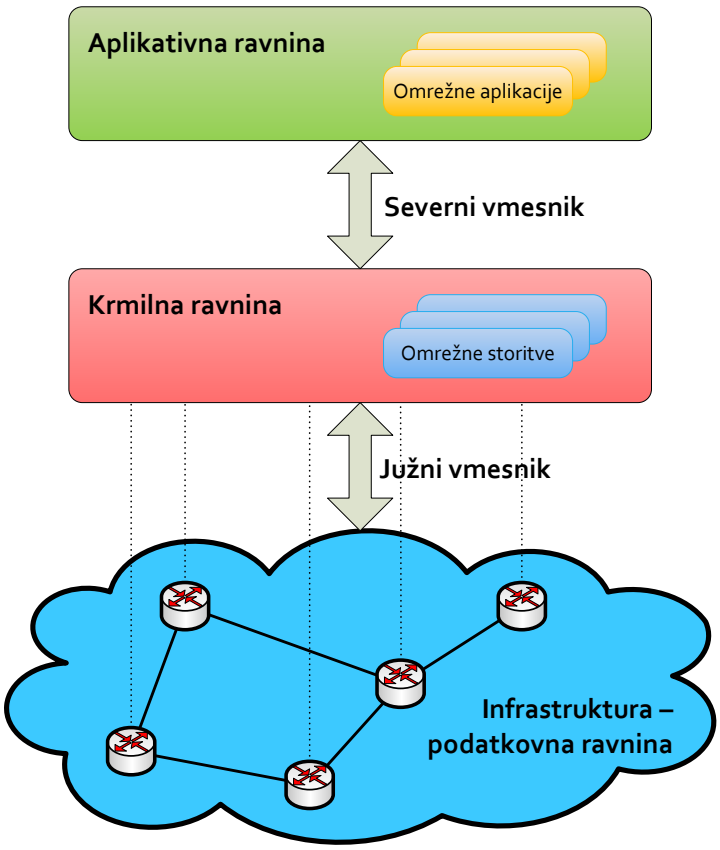
Omrežja SDN so direktno programabilna, centralno upravljana in agilna, programsko konfigurirana ter odprta in neodvisna:

a) Direktno programabilna omrežja, ker je centralizirano krmilno ravnino, ki je ločena od omrežnih naprav, preko višje ležečih omrežnih aplikacij možno direktno programirati.
b) Centralno upravljana in agilna omrežja, ker krmilnik SDN predstavlja (logično) centralno točko za nadzor celotnega omrežja. Tako je celotno omrežje višjim nivojem abstraktno predstavljeno kot eno samo veliko stikalo. Takšna arhitektura omrežnemu administratorju omogoča zelo učinkovito dinamično spreminjanje omrežnih nastavitev.

c) Programsko konfigurirana omrežja, ker omrežja SDN omogočajo, da jih omrežni administratorji upravljajo preko programov, ki jih sami napišejo.

d) Odprta in neodvisna omrežja, ker so vmesniki med ravninami standardni in prosto dostopni oz. odprti. To omogoča poenostavljeno projektiranje in obratovanje omrežja, saj vsi ukazi, namenjeni omrežnim napravam, izvirajo iz enotnega krmilnika in so enotnega formata ne glede na proizvajalca naprave.

Arhitekturo omrežja SDN sestavljalo tri ravnine (Slika 2). Pravila delovanja omrežja postavljajo aplikacije aplikativne ravnine, krmilna ravnina preko omrežnih storitev ta pravila uveljavlja, podatkovna ravnina pa jih z ustreznim posredovanjem podatkovnih paketov izvaja.



Slika 2: Arhitektura SDN

2.2 Krmilna ravnina

Osrednji del omrežja SDN je krmilna ravnina (Slika 2). Njena glavna naloga je izračun in posredovanje ustreznih podatkov v posredovalne tabele podatkovne ravnine, s pomočjo katerih podatkovna ravnina ustrezno obdela in posreduje pakete. Krmilna ravnina ima neke vrste posredniško vlogo med aplikacijami, ki programsko opredelijo delovanje omrežja, in napravami podatkovne ravnine, ki zadane naloge izvajajo. Ta vloga je podobna, kot jo ima operacijski sistem v računalnikih, zato je krmilnik SDN v literaturi pogosto imenovan omrežni operacijski sistem (ang. *Network Operation System*) [11, 15].

Za komunikacijo s spodnjo infrastrukturno oz. podatkovno ravnino krmilna ravnina uporablja t. i. južni vmesnik (ang. *southbound* API), podrobneje predstavljen v poglavju 2.5. Z zgornje strani krmilno ravnino upravlja aplikativna ravnina preko t. i. severnih vmesnikov (ang. *northbound* API). V primeru, da krmilno ravnino sestavlja več krmilnikov, komunikacija med njimi poteka preko t. i. vzhodno-zahodnih vmesnikov (ang. *east/westbound* APIs) [15].

Krmilna ravnina SDN z več krmilniki SDN je logično centralizirana. To pomeni, da se fizično distribuirane krmilne entitete navzven predstavljajo kot enovit sistem. Te entitete se lahko dinamično vključujejo ali izključujejo glede na potrebe, ki izhajajo iz obremenitev v omrežju, ali pa so ves čas vključene za potrebe redundance. Na ta način se zagotavljajo ustrezna zmogljivost, zanesljivost in razširljivost. Redundantne krmilne entitete so običajno tudi geografsko ločene, s čimer zagotovimo t. i. geo-redundanco. Logično centraliziran krmilnik SDN se tipično izvaja v navideznem oz. virtualnem okolju na splošnonamenski strojni opremi v enem ali več podatkovnih središčih (ang. *data center*).

Prednosti centraliziranega krmiljenja v primerjavi s klasičnim, distribuiranim načinom krmiljenja omrežij so [15]:

- enostavnejše spreminjanje omrežnih nastavitev, pri čimer se močno zmanjša verjetnost vnosa administratorskih napak;
- avtonomen in avtomatičen odziv programa krmilnika na spremembe stanj v omrežju in
- enostavnejši razvoj novih naprednih funkcij in storitev, ki jih omogoča centralizirana kontrolna logika z globalnim zavedanjem vseh stanj v omrežju.

Centralizirano napravo omrežja SDN, ki izvaja nalogo krmilne ravnine SDN, imenujemo, krmilnik SDN (ang. SDN *controller*). Predstavniki krmilnikov SDN so: POX [41], NOX [12, 37], Ryu [36], Floodlight [40], Beacon [8], ONOS [34], OpenDaylight [35] in drugi.

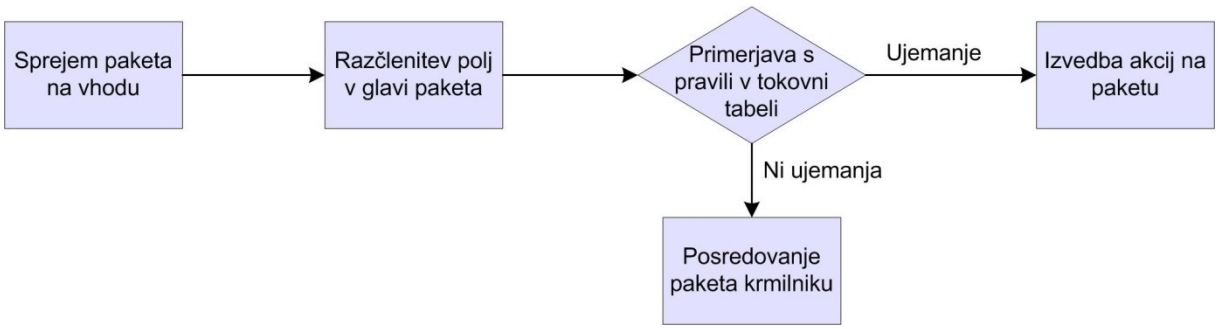
2.3 Podatkovna ravnina

Skupek povezanih naprav, ki izvajajo posredovanje podatkovnih paketov, sestavlja podatkovno ravnino SDN (Slika 2). Naprave podatkovne ravnine za obdelavo oz. posredovanje paketov uporabljajo majhno množico zelo dobro definiranih ukazov, ki jih izvajajo na dohodnih podatkovnih paketih.

Glavni del podatkovne ravnine je ena ali več tokovnih tabel (ang. *flow tables*). V njih se nahajajo omenjeni ukazi v obliki tokovnih pravil (ang. *flow rule*), ki definirajo podatkovni tok (poglavje 2.3). Vsak podatkovni paket oz. njegov del se primerja s pravili v tokovni tabeli, če je primerjava uspešna na več kot enem pravilu, se upošteva prioriteta pravil. Tokovna pravila preko južnega vmesnika (poglavje 2.5) posreduje in sproti posodablja krmilnik SDN.

V osnovi so tokovna pravila sestavljena iz dveh delov: ujemanje oz. primerjalno polje (ang. *match*) in akcijsko polje (ang. *action*), različni protokoli oz. njihove izvedenke, tema dvema poljema dodajajo še razna polja, ki predstavljajo metapodatke, npr. števce, časovnike, prioriteto ipd. Primerjalno polje sestavlja določen bitni vzorec enega ali več predvidenih polj v glavah podatkovnega paketa. Vsi podatkovni paketi, pri katerih so polja v glavi v skladu z določenim vzorcem, se »ujamejo« na to pravilo in zanje velja pripadajoča akcija. V primeru, da se vhodni paket ne ujame na nobeno pravilo, zanj velja implicitna akcija, ki jo tudi določi

krmilnik. Ta akcija je v večini primerov *posreduj na krmilnik*. Slika 3 prikazuje potek obdelave paketa znotraj podatkovne ravnine.



Slika 3: Obdelava paketa na podatkovni ravnini

Tokovna pravila lahko vsebujejo naslednja navodila oz. akcije, ki jih podatkovna ravnina izvaja na dohodnih podatkovnih paketih [15]:

- posreduj na določena vrata,
- pomnoži in posreduj na več vrat,
- prepiši določeno polje v glavi,
- dodaj določeno polje v glavo,
- preveri ujemanje v naslednji tabeli,
- posreduj na krmilnik,
- uporabi in
- odvrzi.

Za posredovanje podatkovnih paketov v omrežjih SDN pravimo, da je tokovno orientirano – poteka na podlagi pravil, ki definirajo podatkovni tok. V klasičnem omrežju je posredovanje podatkovnih paketov destinacijsko orientirano – poteka samo na podlagi destinacijskega oz. ciljnega naslova. Tokovno orientirano posredovanje paketov omogoča učinkovito programabilnost omrežja. Tako je možno v omrežje SDN programsko integrirati napredne omrežne funkcionalnosti, kot so požarni zid, izravnalnik obremenitve (ang. *load balancer*), sistem za zaznavanje vdorov ipd. V klasičnih omrežjih so te funkcije implementirane z namenskimi napravami.

Zaradi univerzalne zasnove se namesto izraza krmiljenje podatkovne ravnine pogosto uporablja izraz programiranje podatkovne ravnine, za tokovna pravila, ki jih narekuje krmilna ravnina, pa izraz programabilna pravila (ang. *programmable rule*).

Naprave omrežja SDN, ki izvajajo naloge podatkovne ravnine SDN, imenujemo (programabilna) stikala SDN (ang. SDN *switch*). Ta so lahko izvedena programsko (npr. v simulacijskih okoljih) ali strojno, kjer se za tokovne tabele običajno uporablja zelo učinkovit, asociativni oz. po vsebini naslovljiv pomnilnik – CAM (ang. *Context Addressed Memory*) [11].

2.4 Podatkovni tok

V prejšnjem poglavju opisana tokovna pravila preko para ujemanje – akcija definirajo t. i. podatkovni tok (ang. *flow*). To je skupina paketov, ki imajo zelo podobne vrednosti v svojih glavah in na popolnoma enak način potujejo po omrežju. Ko krmilnik SDN v tokovno tabelo stikala SDN vpiše določeni podatkovni tok, je za ta tok vzpostavljena povezava preko tega stikala. Kadar je za določen podatkovni tok vzpostavljena povezave preko celotnega omrežja, pravimo, da je za ta tok vzpostavljena tokovna povezava (ang. *flow forwarding path*).

Z uporabo večjega števila polj v glavah paketa za definicijo tokovnega pravila dosežemo finejšo granulacijo podatkovnega prometa. Če npr. pravilo poleg ciljnega naslova končne naprave opredeli tudi vrednost polja ciljnih vrat TCP (ang. *Transmission Control Protocol*) v paketu TCP, je pravilo opredelilo podatkovni tok, namenjen točno določeni aplikaciji na točno določeni končni napravi. Tako je lahko za ta tok skozi omrežje speljana popolnoma lastna povezava, ki jo lahko krmilnik tudi kadarkoli prestavi.

Prvi del tokovnega pravila (ujemanje) je podoben seznamu za kontrolo dostopa – ACL (ang. *Access Control List*) in lahko vsebuje tudi maskirane bite (ang. *wildcards*). Če se podatkovni paket pri primerjavi s prvim delom »ujame« na določeno pravilo, naprava prebere še drugi del pravila in izvede ustrezno akcijo (poglavje 2.3), ki jo pravilo določa. Z uporabo tokovnih pravil se tako zagotovi, da je celoten podatkovni tok podvržen identičnemu obravnavanju na posameznem stikalu SDN.

Krmilna ravnina lahko vsakemu podatkovnemu toku določi svojo prioriteto in ga usmeri po svoji poti proti cilju ter tako izvaja že omenjeno funkcijo izravnavanja obremenitve. Podobno je s pomočjo manipuliranja s podatkovnimi tokovi mogoče realizirati tudi ostale visoko nivojske omrežne storitve oz. aplikacije [1].

Na podlagi razdelitve prometa na podatkovne tokove je možno tudi logično razdeliti omrežje. Pri tem uporabimo več krmilnikov, tako da vsak »vidi« samo svojo abstraktno sliko omrežja. Vsak abstraktni del omrežja je določen s podmnožico podatkovnih tokov, ki jih nadzira dodeljeni krmilnik SDN. Tehnika logičnega razdeljevanja omrežja SDN na podlagi podatkovnih tokov, pasovnih širin, topologije ipd. se imenuje tokovno rezinjenje (ang. *flow slicing*) [1, 15, 24].

2.5 Komunikacija med krmilno in podatkovno ravnino

Komunikacija med krmilno in podatkovno ravnino je v klasičnih omrežjih skrita znotraj naprav in omejena na izvajanje omrežnih protokolov. Tako ima vsak protokol ločeno procesno logiko v krmilni ravnini in večinoma ločeno posredovalno tabelo[1], zato je komunikacija med ravninama izvedena namensko glede na protokol in specifiko naprave.

V tehnologiji SDN ta komunikacija poteka »na daljavo« – po omrežju namesto znotraj naprave. Zato SDN stremi k temu, da bi bila komunikacija čim bolj standardizirana in odprta. Vmesnik, namenjen tej komunikaciji, imenujemo južni vmesnik (Slika 2), ker je to vmesnik, ki ga krmilnik uporablja za komunikacijo s spodnjo (podatkovno) ravnino. Primeri

[1] Sorodni protokoli imajo lahko tudi skupno posredovalno tabelo – npr. različni usmerjevalni protokoli »polnijo« skupno usmerjevalno tabelo.

protokolov, ki definirajo južni vmesnik, so: *OpenFlow* (poglavje 2.6), ForCES (ang. *Forwarding & Control Element Separation*) [27], POF (ang. *Protocol Oblivious Forwarding*) [25].

Za namen podpore heterogenih implementacij stikal obstajajo tudi poizkusi uvedbe dodatne plasti na južnem vmesniku. Takšna plast zagotavlja univerzalno abstrakcijo kakršnekoli implementacije stikala, kar pomeni, da se krmilniku ni potrebno zavedati specifik, ki jih na primer prinašajo različne strojne izvedbe stikal. Poleg tega takšna plast omogoča enostavno menjavo protokola na južnem vmesniku (npr. za novejšo verzijo). Primera dodatnih plasti na južnih vmesnikih sta NetASM (poglavje 5.2.4) in HAL (ang. *OpenFlow Hardware Abstraction Layer*) [18].

Krmilnik SDN lahko protokole za komunikacijo na južnem vmesniku uporablja na dva načina, in sicer proaktivno ali reakcijsko. Najbolj učinkovito delovanje se doseže s kombinacijo obeh načinov komuniciranja [10].

2.5.1 Proaktivni način

O proaktivnem krmiljenju stikal SDN govorimo takrat, ko krmilnik poskrbi, da imajo vsa stikala na poti določenega podatkovnega toka v svojih tokovnih tabelah že vnaprej prisotna tokovna pravila za ta tok. To pomeni, da bo že za prvi paket podatkovnega toka ujemanje pri primerjavi s tokovno tabelo uspešno na vseh stikalih. Med vzpostavljanjem nove omrežne povezave (novi tok) posledično ni potrebna dodatna komunikacija med krmilnikom in stikali SDN.

Proaktivne tokove ustvarijo storitvene aplikacije krmilnika SDN, ki posedujejo informacije o karakteristikah prometa, ki pripada njihovi storitvi. Aplikacije izvedejo krmiljenje tokov običajno takoj, ko se zaženejo. Tokovi pa ostanejo prisotni, dokler se ne zgodi morebitna sprememba v nastavitvah. Takšne proaktivne podatkovne tokove imenujemo tudi statični tokovi (ang. *static flows*). Drugi možen tip proaktivnih tokov je, ko krmilnik spremeni nek tok na osnovi stanja v omrežju, npr. preobremenitve posameznih povezav v omrežju [11].

Prednosti proaktivnih tokov so hitrejše vzpostavljanje povezav za nove podatkovne tokove, ker so le-te vzpostavljene vnaprej, ter manj komunikacije med krmilnikom in stikalom, saj ni pošiljanja paketa proti krmilniku, kar pomeni manjšo obremenitev krmilnika.

Glavna slabost proaktivnih tokov je potreba po zelo velikih tokovnih tabelah na stikalih, saj so tokovni vnosi prisotni v tokovni tabeli, tudi če prometa ni. Posledica tega je, da v primeru uporabe pomnilnika CAM potrebujemo precej dražja stikala, v ostalih primerih pa se pojavijo večje zakasnitve pri posredovanju paketov. Problem pri tem načinu je tudi bistveno bolj zahtevno upravljanje, saj je potrebno vse podatkovne tokove specificirati v konfiguraciji krmilnika [10].

Če bi poskušali zgraditi omrežje SDN s stikali z majhnimi tokovnimi tabelami (cenejši pomnilnik CAM), ki bi jih krmilili proaktivno, bi se morali odpovedati fini granulaciji podatkovnega prometa. Brez take granulacije bi izgubili veliko lepih lastnosti omrežja SDN (poglavje 2.7).

2.5.2 Reakcijski način

Pri reakcijskem načinu krmiljenja prejme krmilnik podatkovni paket od stikala običajno kot posledico neujemanja pri primerjavi paketa s pravili v tokovni tabeli. Paket tako predstavlja nov podatkovni tok, ki ga stikalo še ne pozna. Ko krmilnik prejme paket, ga obdela in stikalo zakrmili z ustreznim tokovnim pravilom. V nadaljevanju bodo do izteka časovnika vsi paketi istega toka obdelani lokalno na stikalu SDN.

Omrežne aplikacije na krmilniku, ki svoje storitvene tokove krmilijo reakcijsko, samo nastavijo parametre storitve znotraj krmilnika. Z upoštevanjem teh parametrov krmilnik pravilno obdela naknadno prejeti paket ter ustrezno zakrmili podatkovno ravnino.

Prednost reakcijskega načina je učinkovita raba tokovne tabele, kar pomeni, da so te tabele lahko bistveno krajše. Slabost tega načina so zakasnitve pri vzpostavljanju povezav za nove podatkovne tokove. Zakasnitev prvih paketov v toku je še občutnejša, če je krmilnik geografsko oddaljen. Če se pojavi veliko število hitro spreminjajočih se tokov, je krmilnik lahko preobremenjen zaradi vsakokratne interakcije med krmilnikom in stikali. Možnost preobremenitve krmilnika se povečuje tudi s povečevanjem omrežja, kar predstavlja problem razširljivosti omrežja SDN (poglavje 2.8). Poseben problem tega načina predstavlja možnost izgube povezljivosti med krmilnikom in stikalom – stikalo v tem primeru deluje zelo omejeno [10].

2.5.3 Hibridni način

Glede na prednosti in slabosti obeh načinov se v praksi priporoča uporabo hibridnega načina [10]. Pri tem omrežni administrator in logika znotraj krmilnika določita, kateri podatkovni tokovi so tisti, ki so stalno prisotni in ključni za delovanje omrežja ter se jih krmili proaktivno, ostale pa se krmili reakcijsko.

2.6 OpenFlow

OpenFlow (OF) velja za prvi (*de facto*) standardni protokol za komunikacijo med ločenima podatkovno in krmilno ravnino pod okriljem fundacije ONF. OF se kot najpomembnejši protokol na južnem vmesniku arhitekture SDN (poglavje 2.1) tudi najpogosteje uporablja v implementacijah SDN.

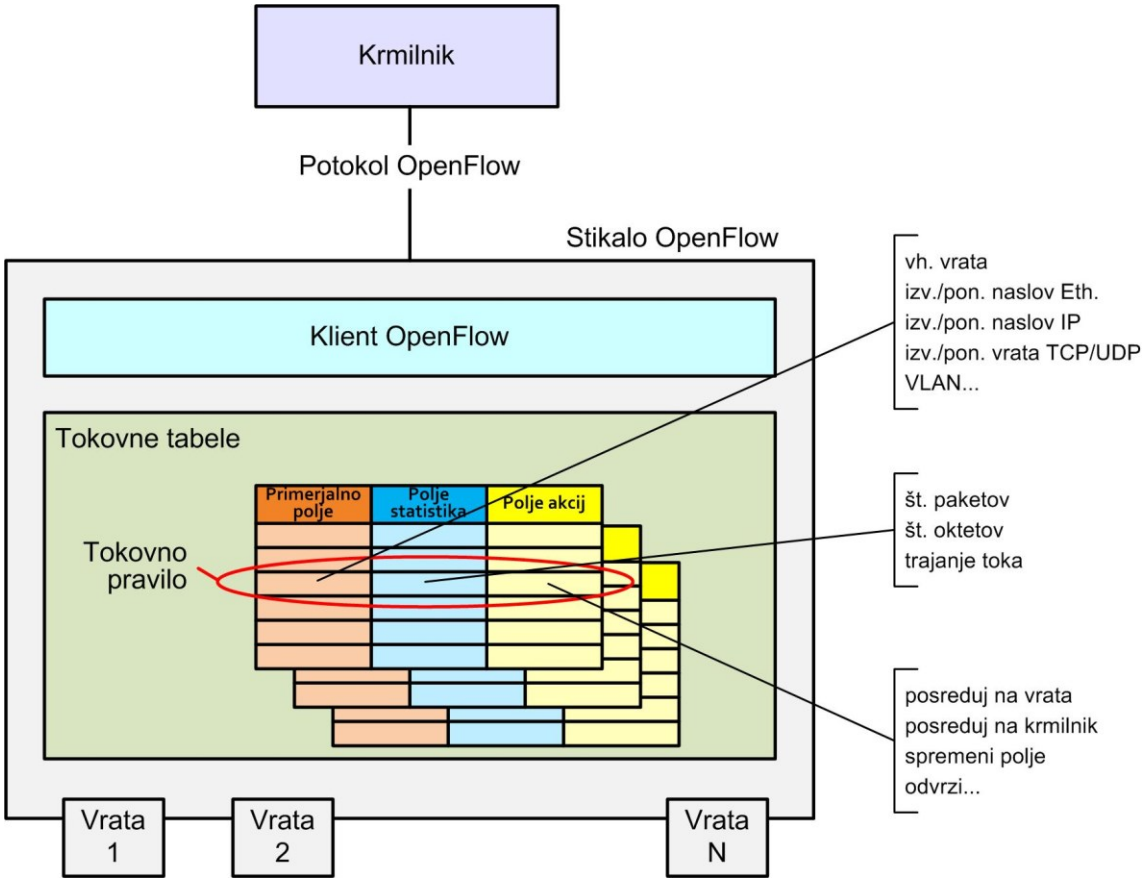
OF se nenehno razvija, zato je specificiranih že več verzij. Novejše verzije omogočajo naprednejše manipulacije s podatkovnimi paketi, poskušajo optimizirati komunikacijo in podpirajo večji nabor omrežnih protokolov, kot na primer IPv6 (ang. *Internet Protocol version 6*) in MPLS (ang. *Multi-Protocol Label Switching*). V nadaljevanju je podan pregled osnovnih konceptov OF, ki so skupni vsem verzijam standarda, podrobna razdelava standarda po posameznih verzijah je na voljo npr. v [5] ali [11].

Omrežje SDN deluje po standardu OF, če vsebuje: krmilnik OF, eno ali več stikal OF in varno (šifrirano) povezavo med njima, ki služi za prenos sporočil protokola OF (Slika 4).

Krmilnik OF krmili stikala OF s pomočjo tokovnih pravil po postopku, opisanem v poglavjih 2.3 – 2.5. V nadaljevanju je zato večji poudarek na opisu stikala in protokola OF.

2.6.1 Stikalo *OpenFlow*

Znotraj stikala OF se odvisno od verzije nahaja ena ali več tokovnih tabel s tokovnimi pravili. Tokovno pravilo v osnovni verziji standarda OF (1.0) sestavljajo tri polja: primerjalno polje, polje števcev oz. statistika in polje akcij (Slika 4).



Slika 4: Arhitektura OpenFlow

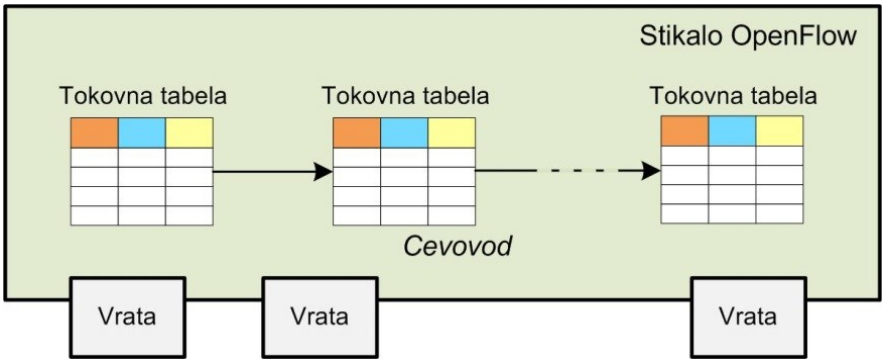
Primerjalna polja znotraj tokovnih pravil določajo pogoj, pod katerim bo izvedena pripadajoča akcija. Osnovni standard OF predvideva naslednje podatke, povezane z vhodnim paketom, ki se lahko uporabljajo kot primerjalni pogoj:

- vhodna vrata prejetega paketa,
- izvorni naslov Ethernet,
- ciljni naslov Ethernet,
- tip okvirja Ethernet,
- značka VLAN (ang. *Virtual Local Area Network*),
- prioriteta VLAN,
- izvorni naslov IP (ang. *Internet Protocol*),
- ciljni naslov IP,

- številka protokola IP,
- biti IP ToS (ang. *Type of Service*),
- izvorna vrata TCP/UDP (ang. *User Datagram Protocol*) in
- ciljna vrata TCP/UDP.

Pogoj primerjalnega polja sestavljajo eden ali več od naštetih podatkov, pri čemer se lahko upošteva celotna vrednost podatka ali samo del vsebine, ki je določen preko maskirnih bitov (ang. *wildcards*).

Nabor podprtih podatkov za primerjanje se z vsako novo verzijo standarda OF močno poveča. Po novem je seznam razdeljen na več razredov in v najnovejšem standardu samo spisek osnovnega razreda vsebuje 45 postavk [33]. Poleg na novo podprtih polj v glavah raznoraznih omrežnih protokolov so v teh seznamih tudi metapodatki, ki se uporabljajo kot navodila pri obdelavi paketa v cevovodu tokovnih tabel (Slika 5).



Slika 5: Cevovod tokovnih tabel znotraj stikala OpenFlow

Polje števcev je namenjeno zbiranju statistike o posameznem podatkovnem toku. Števci beležijo število prejetih paketov, število prejetih oktetov ter trajanje podatkovnega toka.

Akcijsko polje podaja navodilo, kaj naj se s paketom zgodi v primeru uspešne primerjave s primerjalnim poljem – torej v primeru ujemanja. Običajne akcije so *posreduj*, *spremeni polje* in *odvrzi*, širši spisek akcij je podan že v poglavju 2.3. Izvede se lahko tudi več akcij hkrati, npr. sprememba polja v glavi paketa in posredovanje na izhodna vrata. V kolikor se paket ne ujame z nobenim primerjalnim poljem znotraj pravil v celotni tokovni tabeli, se zanj izvede akcija, definirana v posebnem pravilu, imenovanem *table-miss* (Slika 3). Najpogosteje je to akcija *posreduj na krmilnik*, včasih tudi *odvrzi*, lahko pa tudi kaj tretjega.

Kot že omenjeno, je programska oprema krmilnika odgovorna za manipulacijo s tokovnimi tabelami na stikalih. Z vpisovanjem in brisanjem tokovnih pravil krmilnik lahko spreminja obnašanje stikala in tako vpliva na posredovanje prometa. Standard OF definira protokol, ki krmilniku omogoča, da posreduje svoje zahteve stikalom preko varne povezave.

2.6.2 Sporočila protokola *OpenFlow*

Protokol OF predvideva tri tipe sporočil [33, 11], in sicer krmilnik proti stikalu, asinhrona in simetrična sporočila:

- Krmilnik proti stikalu (ang. *controller-to-switch*) je tip sporočil, kjer komunikacijo začne krmilnik. Uporablja se za krmiljenje in preverjanje stanj stikala, kar vključuje poizvedovanje krmilnika o funkcijah, ki jih stikalo podpira, posredovanje konfiguracije, programiranje stikala in zajemanje informacij.

- Asinhrona sporočila so sporočila, pri katerih komunikacijo začne stikalo OF brez povpraševanja s strani krmilnika. Uporabljajo se za obveščanje krmilnika o novem podatkovnem toku, spremembi stanja stikala in sporočanje morebitnih napak.

- Pri simetričnih sporočilih komunikacijo začne krmilnik ali stikalo brez povpraševanja z druge strani. Primera tega tipa sta sporočili *hello* in *echo*, ki se uporabljata pri vzpostavljanju in preverjanju povezljivosti povezave OF.

Standard predvideva več sporočil, izmed katerih vsako spada pod enega od naštetih tipov. Podrobna razlaga vseh sporočil in poteka komunikacije je na voljo v [33], [42] ali [11]. Za potrebe tega dela pa so tu podrobneje predstavljena tri osnovna sporočila ter sporočilo, namenjeno eksperimentiranju:

- Sporočilo *PACKET-IN* je asinhronega tipa. Uporablja se za obveščanje krmilnika o novem podatkovnem toku. Vsebuje osnovne metapodatke (identifikacijska številka stikala, številka vhodnih vrat paketa ipd.) ter običajno celoten paket. Če stikalo omogoča vmesno pomnjenje (ang. *buffering*), celotne vsebine paketa ni potrebno pošiljati, temveč samo prvi del paketa, ki vsebuje glave protokolov, npr. prvih 128 oktetov.

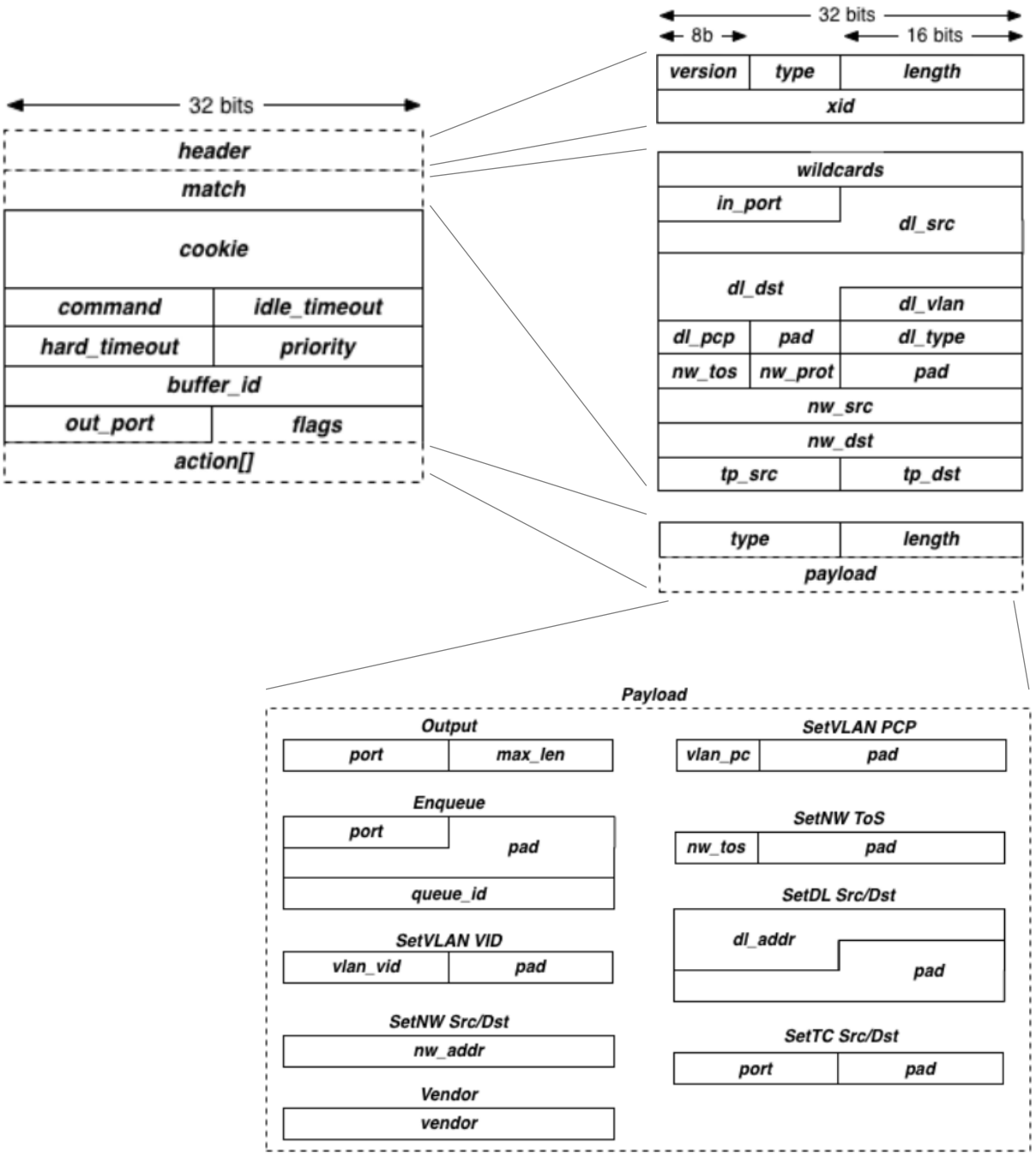
- Sporočilo *FLOW_MOD* je tipa krmilnik proti stikalu. Uporablja se za vpisovanje ali spreminjanje pravila znotraj tokovne tabele izbranega stikala. Krmilnik ga pošlje samoiniciativno (proaktivni način krmiljenja – poglavje 2.5.1) ali pa kot odgovor na sporočilo *PACKET-IN* (reakcijski način krmiljenja – poglavje 2.5.2). Dodatna možnost je pošiljanje kot odgovor na sporočilo o izteku časovnika za posamezno tokovno pravilo ali kadarkoli logika krmilnika to zahteva. Če stikalo omogoča vmesno pomnjenje, omenjeno v prejšnji alineji, se po vpisu novega tokovnega pravila posreduje tudi paket iz medpomnilnika, v nasprotnem primeru pa krmilnik uporabi še sporočilo *PACKET-OUT*.

- Sporočilo *PACKET-OUT* je tipa krmilnik proti stikalu. Krmilnik ga uporablja za vračanje prejetega paketa znotraj sporočila *PACKET-IN* v primeru, da je prejel celotni paket.

- Sporočilo *VENDOR*[2] je simetričnega tipa. Uporablja se za pošiljanje nestandardnih sporočil, definiranih s strani proizvajalca oz. za eksperimentiranje.

Slika 6 prikazuje primer formata sporočila OF.

[2] Od verzije OF 1.1 naprej se to sporočilo imenuje *EXPERIMENTER*.



Slika 6: Format paketa FLOW_MOD (vir [42])

Razlaga posameznih polj s Slike 6 ter formati ostalih sporočil OF so na voljo v [33] ali [42].

2.7 Prednosti in motivi za gradnjo omrežij SDN

Arhitektura klasičnega omrežja je bila zasnovana v času, ko so komunikacijske naprave zelo pogosto odpovedovale. Glavno vodilo zasnove je bilo zato odpornost na izpade, kar je vodilo v visoko distribuiran sistem, ki v omrežjih prevladuje še danes. Da distribuiran omrežni sistema učinkovito izvaja svojo nalogo omogočanja hitre komunikacije brez izpadov, so potrebni kompleksni omrežni protokoli. Upravljanje teh protokolov na velikem omrežju je zelo zahtevna naloga.

Že tako zahtevno upravljanje dodatno oteži dejstvo, da ne obstaja univerzalni vmesnik za upravljanje. Vsak proizvajalec omrežne opreme tako ponuja svoj upravljavski vmesnik. Poleg konfiguriranja vsak proizvajalec ponuja tudi svojo lastno implementacijo protokolov ter strojne in programske opreme. Takšne popolnoma namenske in avtonomne omrežne naprave ter v celoti razvite v okviru enega proizvajalca, seveda niso poceni.

Omrežja, sestavljena iz visokotehnoloških, avtonomnih in popolnoma zaprtih elementov, ni le težko upravljati, še veliko teže (dražje) jih je nadgraditi z novimi funkcijami. Kot alternativa opisanim omrežjem se ponuja omrežna arhitektura SDN, prednosti katere so opisane v naslednjih razdelkih.

2.7.1 Enostavnejša centralizirana krmilna ravnina

V stabilnih (z zelo malo izpadov) okoljih[3] ni potrebe po kompleksnih distribuiranih mehanizmih oz. protokolih, ki omogočajo avtonomno prilagajanje posredovalnih tabel znotraj omrežnih naprav. V takšnih okoljih je bolj smiselna centralna krmilna ravnina, ki ima vedenje o celotni topologiji (vključno z alternativnimi potmi). Takšna arhitektura lahko akcije ob spremembah topologije izvede bistveno bolj učinkovito. Ker je sprememb topologij v stabilnih okoljih zelo malo, se lahko z enostavnim centraliziranim pristopom učinkovito krmili zelo velika omrežja.

2.7.2 Večja prožnost omrežja

Podatkovna ravnina SDN (poglavje 2.3) omogoča vpogled v več polj v glavah paketa do poljubne globine, v nekaterih izvedbah pa kar vpogled v celoten paket za razliko od klasičnega omrežja, pri katerem je posredovanje velike večine paketov izvedeno samo z vpogledom v polja s ciljnimi naslovi. Z globljim vpogledom v vsebino paketa pridobimo bogatejšo množico informacij, s tem omrežje dobi moč, da izvaja nove funkcije. Tako je odpravljena potreba po izvajanju naprednejših omrežnih funkcionalnosti na ločenih napravah. Funkcije, kot so požarni zid, izravnalnik obremenitve, sistem za zaznavanje vdorov ipd., je v omrežja SDN možno enostavno integrirati z ustrezno rabo programabilnih stikal SDN.

2.7.3 Enostavnejše upravljanje

Arhitektura omrežij SDN bistveno izboljšuje zgoraj omenjene težave z upravljanjem, nadzorom in vzdrževanjem klasičnih omrežij [2, 15]. V klasičnem omrežju je npr. pri vzpostavljanju visoko nivojskih omrežnih pravil potrebno konfigurirati vsako napravo omrežja posebej, pri čemer je pogosto potrebna uporaba nizkonivojskih ukazov. Ti niso različni samo od proizvajalca do proizvajalca, temveč pogosto tudi med različnimi verzijami produkta istega proizvajalca. Primer visokonivojskega omrežnega protokola, pri katerem je pravilna konfiguracija še posebej zahtevna, je protokol BGP (ang. *Border Gateway Protocol*) [28].

Poleg osnovnega konfiguriranja so v klasičnih omrežjih običajno potrebni ročni posegi v primerih dinamičnih sprememb v omrežju, kot so npr. spremembe v količini prometa. Univerzalno avtomatizirano upravljanje je za takšne primere v obstoječih omrežjih redko na voljo.

[3] Še posebno dober primer takšnega okolja so omrežja znotraj podatkovnega središča.

SDN bistveno izboljšuje možnosti za razvoj in uporabo naprednih upravljavskih orodij.

2.7.4 Neodvisnost in odprtost sistema ter standardni vmesniki

Ena večjih »bolečin« omrežnih operaterjev je, da nakup omrežne opreme določenega proizvajalca pomeni vezavo na tega proizvajalca pri vseh prihodnji dograditvah omrežja. SDN zato spodbuja uporabo belih škatel (ang. *white box*) za strojno opremo stikal SDN ter uporabo splošnonamenskih strežnikov kot strojno opremo za krmilnik SDN.

Še večji poudarek se daje standardizaciji in odprtosti vmesnikov, tako na severni kot južni strani arhitekture SDN (Slika 2). Samo ob podpori visoko standardiziranih vmesnikov bo možno kombinirati aplikacije različnih proizvajalcev na eni strani ter stikal v obliki belih škatel na drugi strani.

Ne nazadnje se za večino aplikacij krmilne ravnine pričakuje, da bodo na voljo v obliki odprtokodnih rešitev (ang. *open source*).

2.7.5 Hitrejša vpeljava novih storitev

Že večkrat omenjena visoka programabilnost omrežja SDN omogoča vpeljavo novih inovativnih rešitev in svobodno spreminjanje obnašanja omrežja s preprosto menjavo ali dodajanjem aplikacije na krmilniku. To je lahko izvedeno v manj kot eni uri, za podobno nadgradnjo klasičnega omrežja bi potrebovali več tednov.

Dodatna prednost, ki omogoča hitrejše inoviranje, je možnost testiranja novosti na živem omrežju. Eksperimentiranje »v živo« je elegantno izvedljivo s tehniko tokovnega rezinjenja (poglavje 2.4), pri katerem lahko definiramo izbrane podatkovne tokove, za katere veljajo nove nastavitve omrežja. Na vse ostale podatkovne tokove takšen test nima nikakršnega vpliva in tako ni skrbi za morebitne negativne posledice testiranja.

2.7.6 Natančnejši nadzor in večja varnost

Tehnologija SDN s pomočjo fine granulacije podatkovnega prometa (poglavje 2.4) omogoča natančnejši legalni nadzor nad prometom, s tem pa tudi večjo varnost omrežja.

2.7.7 Nižji stroški za operaterja

Vse opisane prednosti skupaj naj bi v praksi prinesle najbolj zaželeno prednost, to je nižja cena v primerjavi s klasičnimi omrežji. Največji prihranek se pričakuje pri strojni opremi zaradi selitve »pameti« iz omrežnih naprav na splošnonamenske strežnike v podatkovnih centrih. Večinski delež prihranka tako zagotovi velika množica omrežnih naprav, ki v SDN ne potrebuje izdatne procesorske moči ter napredne programske opreme za krmilni del. Potrebna procesorska moč bo v SDN centralizirana v splošnonamenskih strežnikih, ki so cenejši od namenske strojne opreme, poleg tega centralizacija skupaj z virtualizacijo prinaša možnost boljše izkoriščenosti procesorske moči, ki je na voljo.

Prihranke pri programski opremi naj bi omogočali tudi standardni vmesniki in odprtost, omenjena zgoraj. Prav tako naj bi precej prihrankov prineslo tudi že omenjeno enostavnejše upravljanje in vzdrževanje omrežja s pomočjo naprednejših orodij, ki naj bi prinesla veliko avtomatiziranih postopkov.

Prispevek k ugodnejši finančni sliki obeta tudi možnost veliko hitrejše vpeljave novih storitev, saj se bodo investicije v nove storitve tako hitreje povrnile.

Trenutno stanje na trgu rešitev SDN kaže, da je investicija v programsko opredeljeno omrežje sicer višja, kot bi bila v enakovredno klasično omrežje, vendar se predvideva, da bo cena vzdrževanja omrežja SDN bistveno nižja. Predvideva se tudi, da bo ob povečanju konkurence in prisotnostjo vedno večjega števila odprtokodnih aplikacij za posamezne dele rešitve tudi investicija v programsko opredeljena omrežja ustrezno nižja.

2.8 Izzivi arhitekture SDN

Kot bodoče izzive oz. trenutne slabosti omrežja SDN se pogosto navaja t. i. »porodne težave«, ki so značilne za vsako novo tehnologijo [11, 15, 31]. Mednje spadajo odsotnost oz. nedokončani standardi, slaba interoperabilnost aplikacij in opreme, pomanjkanje podpore v industriji in majhno število delujočih inštalacij v praksi.

Resnejši očitki trenutnega stanja tehnologije SDN pa so zapisani v nadaljevanju [11, 15, 31]:

- Zakasnitev pri vzpostavljanju novega podatkovnega toka.

  Pri reakcijskem načinu krmiljenja (poglavje 2.5.2) vzpostavitev novega podatkovnega toka povzroča precejšnjo zakasnitev prvega paketa v toku. Problematika je podrobno razdelana v poglavju 3.

- Slaba zmogljivost krmilnika.

  Dobršen del zakasnitve pri vzpostavljanju novega podatkovnega toka predstavlja čas, ki ga krmilnik potrebuje za obdelavo prvega paketa ter za vpis novega pravila v stikalo. Za storitve v realnem času pogosto že ta zakasnitev predstavlja problem. Kadar imamo obsežno omrežje, lahko število poslanih paketov proti krmilniku preseže njegovo zmogljivost. Takrat pride do zasičenja centraliziranega krmilnika in že tako neugodne zakasnitve pri vzpostavljanju tokov, se še močno povečajo. S pomočjo numerične analize [13, 26] je za konkretne karakteristike omrežja možno izračunati potrebno zmogljivost krmilnika SDN.

- Slaba razširljivost krmilnika.

  Omogočiti učinkovito razširljivost je glavna skrb tehnologije SDN že od samega začetka. V prejšnji alineji omenjeno nevarnost zasičenja krmilne ravnine pri povečevanju omrežja ali pojavljanju večje količine novih tokov je potrebno pravočasno zaznati in ustrezno ukrepati. Sofisticirano navidezno krmilno okolje bi moralo nevarnost samodejno zaznati in avtomatsko zagnati nove fizične instance krmilnika. Dodatno težavo glede razširljivosti predstavlja komunikacija za usklajevanje podatkov med instancami krmilnika, če je instanc veliko, že samo usklajevanje lahko prekomerno obremenjuje krmilnik [16].

- Slaba modularnost krmilnika.

  Po eni strani je odraz slabe modularnosti odsotnost standardnega protokola za komunikacijo med instancami krmilnika, tj. med vzhodno-zahodnimi vmesniki (poglavje 2.2). Po drugi strani se slaba modularnost kaže v odsotnosti standardnega

vmesnika med aplikacijami krmilnika, tako da je do v uvodu omenjenega dinamičnega nalaganja aplikacij po vzoru pametnih telefonov še daleč.

- Visoka razpoložljivost ni zagotovljena.

  Centralni krmilnik omrežja SDN predstavlja eno samo točko odpovedi (ang. *single point of failure*). Za učinkovito odpravo nevarnosti izpada omrežja zaradi izpada ene fizične instance krmilnika bo potrebno definirati standardni postopek za preklop in prevzemanje nadzora nad stikali med instancami krmilnika. Najbolj obetajoči predlog na tem področju je vezava vsakega od stikal na več krmilnikov [20]. Dodatna prednost te rešitve je, da se hkrati varuje tudi morebitni izpad povezave med stikalom in krmilnikom.

- Odpornost omrežja.

  Izpada katere izmed povezav v omrežju SDN zazna stikalo, ki o tem obvesti krmilnik. Krmilnik nato izračuna nadomestno pot in osveži vsa tokovna pravila za prizadete podatkovne tokove na vseh relevantnih stikalih. Takšna rekonstrukcija povezljivosti lahko zahteva precej časa in povzroči nedopustne zakasnitve.

- Omejena kapaciteta tokovnih tabel na stikalih.

  Večina strojno izvedenih stikal SDN za tokovne tabele uporablja pomnilnike CAM, ki omogočajo hkratno primerjavo vseh pravil v tabeli. Problem teh pomnilnikov je, da so zelo dragi in energijsko potratni. Posledično proizvajalci v stikala vgrajujejo majhne velikosti pomnilnikov CAM. Zaradi tega se lahko hitro zgodi, da v tokovni tabeli stikala zmanjka prostora za vsa potrebna tokovna pravila.

- Varnost omrežja.

  Tako kot pri kateremkoli modernem informacijskem sistemu je tudi pri omrežju SDN potrebno veliko pozornosti posvetiti varnosti. Zaradi centraliziranega krmiljenja je lahko potencialna škoda ob morebitnem nepooblaščenem posegu v primerjavi s klasičnim omrežjem bistveno večja. Šibke varnostne točke omrežja SDN, ki zahtevajo poglobljeno obravnavo:
  - Nevarnost vsiljevanja lažnih tokovnih pravil v podatkovno ravnino.
  - Ranljivosti stikal, ki bi jih morebitni napadalec lahko izkoristil za napad.
  - Morebiten uspešen napad na centraliziran krmilnik ali upravljavski sistem bi napadalcu omogočil nadzor nad celotnim omrežjem.
  - Pojav okvarjenega ali zlonamernega krmilnika ali aplikacije znotraj krmilnika lahko ogrozi delovanje celotnega omrežja.
  - Odsotnost orodij in postopkov za forenzično preiskavo in sanacijo napadenega omrežja.

3 METODE KRMILJENJA STIKAL SDN

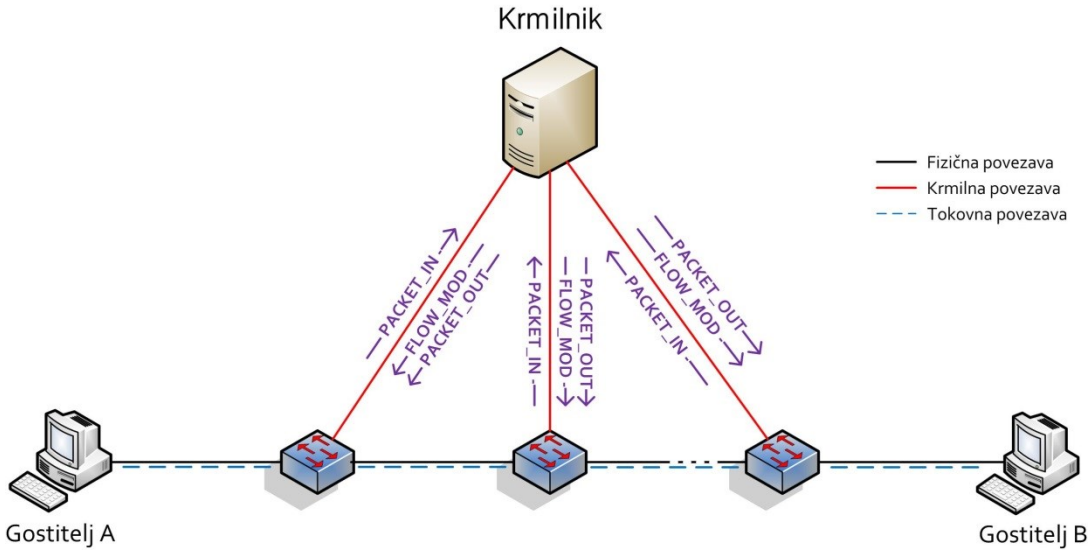
Med izzivi tehnologije SDN, zapisanimi v poglavju 2.8, v literaturi [1, 13, 15], najbolj izstopata nezadostna zmogljivost in razširljivost krmilnika. Zaskrbljenost se nanaša na reakcijski način krmiljenja, saj obstaja resna nevarnost, da bi v primeru večjih omrežij ali več kratkotrajnih tokov, krmiljeni tokovi občutili prevelike zakasnitve pri njihovem vzpostavljanju. To še posebej velja za podatkovne tokove aplikacij, ki zahtevajo odzivnost v realnem času.

Reakcijsko krmiljenje oz. vzpostavljanje povezljivosti krmilnik lahko izvaja z uporabo različnih metod. V tem poglavju je opisana osnovna metoda reakcijskega krmiljenja (Metoda I) ter njena enostavna izboljšava (Metoda II). Ker tudi izboljšana metoda ne odpravlja zgoraj omenjenih glavnih slabosti reakcijskega krmiljenja, je v poglavju 4 opisana povsem nova metoda reakcijskega krmiljenja (Metoda III).

Predstavljene metode lahko uporablja katerikoli protokol na južnem vmesniku arhitekture SDN. Za namen predstavitve v nadaljevanju uporabljamo termine, kot jih predvideva standard *OpenFlow*.

3.1 Osnovna metoda krmiljenja

Potek reakcijskega vzpostavljanja nove tokovne povezave skozi omrežje SDN prikazuje Slika 7. Vidimo, da je za vzpostavitev tokovne povezave med gostiteljem A in gostiteljem B potrebno precej interakcije s krmilnikom. Ta interakcija povzroča zakasnitve pri vzpostavljanju tokovne povezave.



Slika 7: Osnovno reakcijsko krmiljenje (Metoda I)

S Slike 7 lahko razberemo dogajanje ob vzpostavljanju tokovne povezave po Metodi I: ko prvi paket novega podatkovnega toka potuje od izvora proti cilju, vsako stikalo na tej poti ugotovi, da za ta paket ni ujemanja v njegovi tokovni tabeli, zato ga pošlje krmilniku

(sporočilo *PACKET_IN*). Da se vzpostavi celotna tokovna pot oz. sled preko omrežja, krmilnik isti paket dobi v obdelavo tolikokrat, kolikor je stikal na poti. Hkrati ta paket tudi tolikokrat pošlje nazaj posameznemu stikalu (sporočilo *PACKET_OUT*). V enakem obsegu tudi sproti generira krmilna sporočila oz. pravila za posredovanje novega podatkovnega toka, ki predstavljajo vnos v tokovno tabelo na stikalih (sporočilo *FLOW_MOD*). Število vseh prenosov sporočil na relaciji stikalo–krmilnik za dolžino poti oz. za število stikal na poti tokovne povezave, N, lahko izračunamo po enačbi:

$$M_{SUM\_1} = 3 \times N \quad (1.0)$$

Zakasnitev prvega paketa novega podatkovnega toka na vsakem stikalu je vsota časa, potrebnega za prenos paketa na krmilnik (T_{PACKET_IN}), časa obdelave paketa na krmilniku (T_C), časa, potrebnega za sestavo, oddajo in prenos tokovnega pravila (T_{FLOW_MOD}), časa vračanja paketa na stikalo (T_{PACKET_OUT})[4] ter časa posredovanja paketa na stikalu skupaj s časom prenosa paketa do naslednjega stikala (T_{FWD}). Dobljeno vsoto pomnožimo še s številom stikal na poti tokovne povezave (N). Tako dobimo enačbo za skupno zakasnitev prvega paketa novega podatkovnega toka za primer s Slike 7 (T_{SUM_1}):

$$T_{SUM\_1} = (T_{PACKET\_IN} + T_C + T_{FLOW\_MOD} + T_{PACKET\_OUT} + T_{FWD}) \times N \quad (1.1)$$

Za stikala se zahteva, da svojo nalogo obdelave oz. posredovanja paketov izvedejo tako hitro, kolikor hitro paketi prihajajo po fizični povezavi, tudi če je ta polno obremenjena. Torej posredovanje s hitrostjo žice (ang. *line rate*). Skupni čas posredovanja paketa na stikalu in prenosa po fizični povezavi (T_{FWD}) je zato bistveno krajši od ostalih časov. Poleg tega se analiza osredotoča na obremenitve krmilnika, ki je kot centralna točka glede tega bolj občutljiva kot stikala. Zaradi majhnega vpliva in nerelevantnosti čas T_{FWD} v izračunu zanemarimo. Tako dobimo novo enačbo:

$$T_{SUM\_1} = (T_{PACKET\_IN} + T_C + T_{FLOW\_MOD} + T_{PACKET\_OUT}) \times N \quad (1.2)$$

Časi, ki predstavljajo zakasnitve pri komunikaciji med stikalom in krmilnikom (T_{PACKET_IN}, T_{FLOW_MOD} in T_{PACKET_OUT}), poleg prenosa po fizični povezavi zajemajo tudi čas sprejema oz. oddaje na krmilniku ter morebitno čakanje sporočila v vhodni vrsti, česar pa ni mogoče zanemariti. Vseeno lahko izračun poenostavimo tako, da zanemarimo razlike med temi tremi časi in vse tri predstavimo kot čas za prenos sporočila (T_M). Tako dobimo končno enačbo za izračun zakasnitve prvega paketa novega podatkovnega toka:

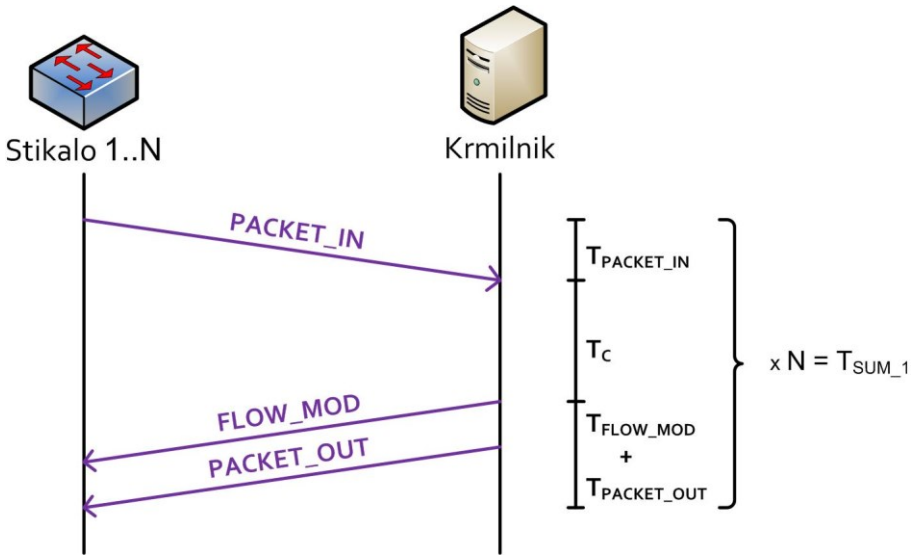
$$T_{SUM\_1} = (T_C + 3 \times T_M) \times N \quad (1.3)$$

oziroma ob upoštevanju enačbe 1.0:

$$T_{SUM\_1} = N \times T_C + M_{SUM\_1} \times T_M \quad (1.4)$$

Slika 8 prikazuje še časovni potek reakcijskega vzpostavljanja tokovne povezave.

[4] Tu ni upoštevana možnost hranjenja paketa v vmesnem pomnilniku na stikalu (poglavje 2.6.2).



Slika 8: Časovni potek reakcijskega krmiljenja (Metoda I)

Zmanjšanje skupne zakasnitve prvega paketa novega podatkovnega toka skozi omrežje T_{SUM_1} je možno doseči s povečevanjem zmogljivosti krmilnika (manjša T_C in T_M) ali pa z zmanjšanjem potrebnega števila izmenjanih sporočil pri vzpostavljanju tokovne povezave. Zmogljivost krmilnika lahko povečamo z uporabo zmogljivejše strojne opreme, z vključevanjem večjega števila fizičnih krmilnikov (poglavje 2.2) ali z izbiro optimalnejšega krmilnika, npr. Beacon [8]. Za zmanjšanje števila izmenjanih sporočil pri vzpostavljanju tokovne povezave lahko na primer poskrbimo s spremembo logike krmilnika (poglavje 3.2).

3.2 Izboljšana metoda krmiljenja

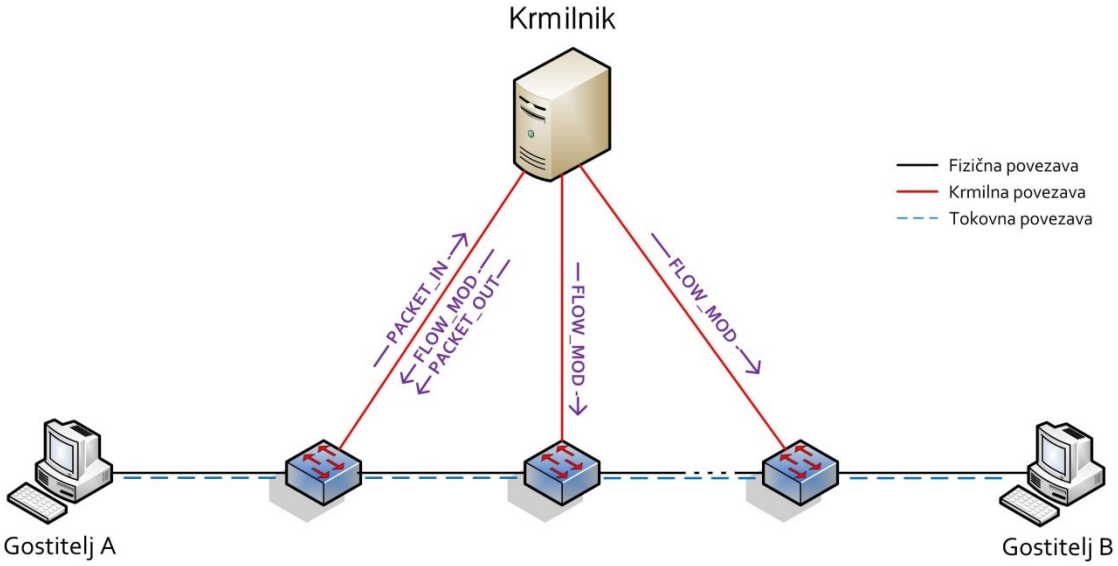
V prejšnjem poglavju predstavljena metoda reakcijskega krmiljenja je osnovna različica, pri kateri je izvedba programa na krmilniku najenostavnejša. Krmilnik pri tej metodi ne izkorišča možnosti globalnega pogleda na omrežje, ki je sicer ena od glavnih prednosti tehnologije SDN (poglavje 2.7.1).

Glede na Metodo I (poglavje 3.1) je v primeru reakcijskega krmiljenja z zavedanjem topologije (Metoda II) izvedena enostavna izboljšava logike krmilnika. Krmilnik uporablja naprednejšo različico omrežne aplikacije. Ta omogoča poizvedovanje po topologiji omrežja in s tem izkoristi globalni pogled na omrežje. Krmilnik si tako ustvari lastno sliko topologije omrežja, kar mu omogoča, da lahko učinkoviteje krmili podatkovno ravnino. Posledično dobimo boljše rezultate, tako glede obremenjenosti krmilnika kot tudi glede zakasnitve prvega paketa novega podatkovnega toka.

Pri Metodi II prvo stikalo na poti paketa, ki predstavlja nov podatkovni tok, ugotovi, da za ta paket ni ujemanja v tokovni tabeli in ga pošlje v obdelavo krmilniku (*PACKET_IN*). Krmilnik s pomočjo informacij o topologiji omrežja identificira vsa stikala na poti paketa in jim tudi pošlje krmilna sporočila za vzpostavitev novega podatkovnega toka (N x *FLOW_MOD*). Nazadnje krmilnik vrne paket prvemu stikalu[5] (*PACKET_OUT*). Paket nato nadaljuje pot po

[5] Krmilnik lahko paket vrne tudi zadnjemu stikalu na poti, vendar to ne vpliva na naš izračun zakasnitve, saj smo čas, potreben za posredovanje paketov na podatkovni ravnini, zanemarili (poglavje 3.1).

že vzpostavljeni povezavi vse do ciljnega gostitelja. Slika 9 prikazuje vzpostavitev tokovne povezave med gostiteljem A in gostiteljem B za krmiljenje po Metodi II.



Slika 9: Izboljšano reakcijsko krmiljenje (Metoda II)

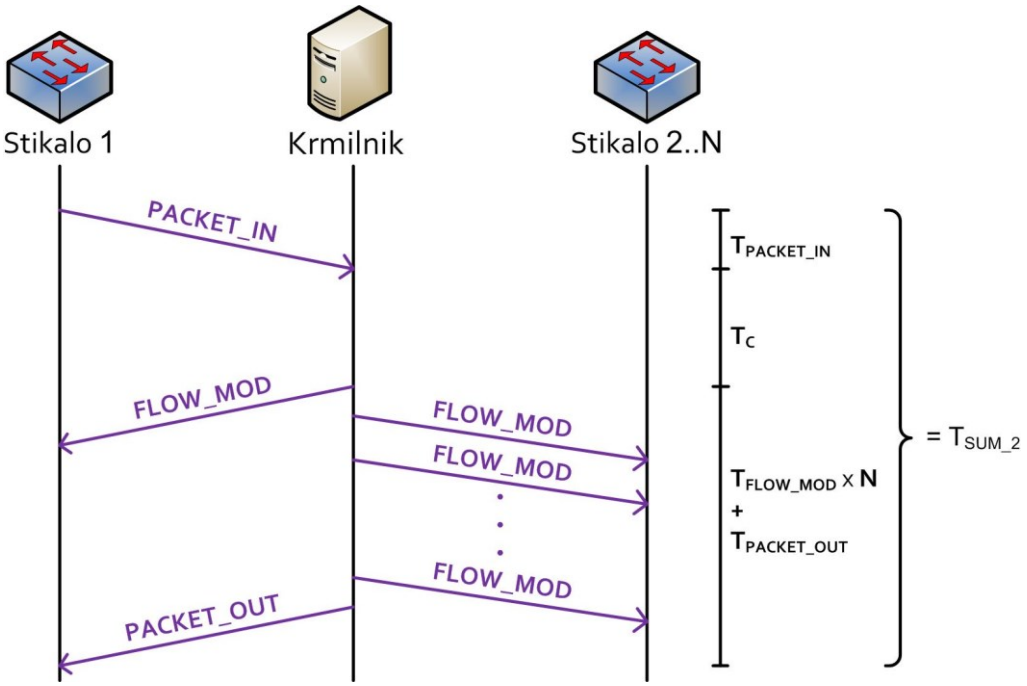
Število vseh prenosov sporočil na relaciji stikalo–krmilnik za dolžino poti N stikal za Metodo II lahko izračunamo po enačbi:

$$M_{SUM\_2} = N + 2 \qquad (2.0)$$

Razlika v številu sporočil med Metodo I in Metodo II je tako:

$$\Delta M_{12} = M_{SUM\_1} - M_{SUM\_2} = 2N - 2 \qquad (2.1)$$

Slika 10 prikazuje časovni potek reakcijskega vzpostavljanja tokovne povezave za Metodo II.



Slika 10: Časovni potek izboljšanega reakcijskega krmiljenja (Metoda II)

Skupno zakasnitev prvega paketa novega podatkovnega toka za dolžino poti N, izraženo s časom obdelave na krmilniku (T_C)[6] ter časom, potrebnim za prenos posameznega sporočila (T_M), za Metodo II, lahko izračunamo po enačbi:

$$T_{SUM\_2} = T_C + M_{SUM\_2} \times T_M = T_C + (N + 2) \times T_M \qquad (2.2)$$

S to metodo se tako zakasnitev zmanjša glede na Metodo I za:

$$\Delta T_{12} = T_{SUM\_1} - T_{SUM\_2} = N \times T_C + 3 \times N \times T_M - T_C - N \times T_M - 2 \times T_M =$$

$$= (N - 1) \times (T_C + 2 \times T_M) \qquad (2.3)$$

Vidimo, da je Metoda II precej optimalnejša od Metode I. Vseeno se ohranja linearna odvisnost obeh izračunanih karakteristik od števila stikal na poti. Da bi odpravili na začetku poglavja 3 omenjene glavne slabosti reakcijskega krmiljenja, bi potrebovali metodo krmiljenja, pri kateri bi bilo tako število sporočil kakor tudi zakasnitev prvega paketa neodvisno od dolžine poti. Takšnemu idealu se zelo približa metoda, predstavljena v naslednjem poglavju.

[6] Čas obdelave na krmilniku je pri tej metodi seveda nekoliko daljši kot pri Metodi I, vendar je ta razlika, v primerjavi z razliko časa generiranja in pošiljanja paketov zanemarljivo majhna.

4 NOVA METODA – RAZPRŠENO REAKCIJSKO KRMILJENJE

Poleg v prejšnjem poglavju opisane modifikacije krmiljenja so bile predlagane še druge tehnike, ki zmanjšujejo potrebo po interakciji med krmilnikom in stikali. Nekateri predlagajo hierarhične arhitekture krmilnikov, pri katerih za krmiljenje večine tokov poskrbi lokalni posredovalni (ang. *proxy*) krmilnik, npr. *DevoFlow* [7]. Spet drugi predlagajo, da robno stikalo v vse podatkovne pakete dodaja krajše programe, ki vsebujejo navodila za njihovo posredovanje na vseh ostalih stikalih na poti, npr.: *Active networks* [21], *Source flow* [6] ali *Tiny Packet Programs* [14]. Takšna tehnika omogoča, da krmilnik vzpostavi celotno tokovno povezavo tako, da komunicira samo s prvim stikalom na poti.

Povsem nov pristop za zmanjšanje potrebe po omenjeni interakciji, ki vpeljuje uporabo razpršenega (ang. *multicast*) oddajanja krmilnih sporočil, je predlagan in podrobno predstavljen v nadaljevanju. Predlagana nova metoda sicer na prvi pogled daje vtis kompleksnosti, vendar je v primerjavi z omenjenimi tehnikami enostavna.

Predlagamo torej spremembo poljubnega protokola na južnem vmesniku, od katerega se pričakuje reakcijsko vzpostavljanje podatkovnega toka oz. krmiljenje. V našem primeru je bil kot osnova izbran protokol *OpenFlow*. Razlaga nove metode in predlaganih modifikacij tako slonijo na tem protokolu.

4.1 Izhodišča nove metode

Izhodišča metode z razpršenim reakcijskim krmiljenjem so:

- Velika večina akcij, ki jih stikala znotraj omrežja SDN izvajajo nad podatkovnimi tokovi, so enostavno posredovanje na izhodna vrata. Akcije, kot so: *prepiši polje*, *dodaj polje*, *odvrzi* ipd. (poglavje 2.3), se večinoma izvajajo samo na robu omrežja[7].
- Akcije na robnih stikalih, ki paketom spreminjajo kakšno polje v glavi (npr. VLAN značko), so skupne vsem podatkovnim tokovom, ki sestavljajo določeno uporabniško storitev.
- Ker je v tokovnih tabelah znotraj stikala zelo veliko vnosov, množica različnih akcij pa je majhna, se večina akcij velikokrat ponovi.
- Glave paketov se razen na robnih stikalih ne spreminjajo, zato so primerjalna polja za določen tok po celotni poti popolnoma enaka.

Opisane enakosti akcijskih polj znotraj tokovnih tabel in primerjalnih polj vzdolž tokovne poti so spodbudile razmišljanja o možnosti vzpostavljanja tokovnih povezav preko celotnega oz.

[7] Obstajajo tudi izjeme, ko se kakšno od polj v glavi paketa lahko spremeni na vsakem stikalu, npr. protokol MPLS.

velikega dela omrežja z enim samim krmilnim sporočilom. Takšno krmiljenje bi občutno skrajšalo problematično zakasnitev pri vzpostavljanju tokovnih povezav.

4.2 Definicije novih pojmov

Na tem mestu je podana definicija novih pojmov, ki so potrebni pri obravnavi nove metode krmiljenja.

- Krmilno sporočilo *MCAST_RULE.*

  Za potrebe sočasnega krmiljenja tokovne povezave po celotni poti definiramo novo večponorno (ang. *multicast*) krmilno sporočilo, *MCAST_RULE*. To sporočilo stikalu pove, katero od predhodno vzpostavljenih tokovnih povezav naj uporabi kot nadomestno povezavo. Tako kot sporočilo *FLOW_MOD* (poglavje 2.6.2) je tudi to sporočilo tipa krmilnik proti stikalu. Za razliko od sporočila *FLOW_MOD* sporočilo *MCAST_RULE* vpisovanje ali spreminjanje pravila lahko opravi na več kot enem stikalu.

- Nadomestna sled.

  Povezava, katere ponovno uporabo določa sporočilo *MCAST_RULE*, v splošnem ni pouporabljena v celoti, temveč samo njen del. Pouporabljen del poti zato imenujemo nadomestna sled (ang. *alternative trail*). Nova pot tokovne povezava je tako sestavljena iz enega ali več delčkov obstoječih tokovnih povezav – sledi. Krmilnik vsako od nadomestnih sledi zakrmili z enim samim sporočilom *MCAST_RULE*.

- Tabela večponornih pravil.

  Pravilo, ki ga pošlje krmilnik preko sporočila *MCAST_RULE*, se na stikalu shrani v novo definirano tabelo, imenovano tabela večponornih pravil (ang. *Multicast rule table*). Tabela vsebuje podobne podatke kot tokovna tabela, le da namesto stolpca akcij vsebuje kazalce na nadomestne sledi v tokovni tabeli (Slika 13).

- Cepitveno vozlišče.

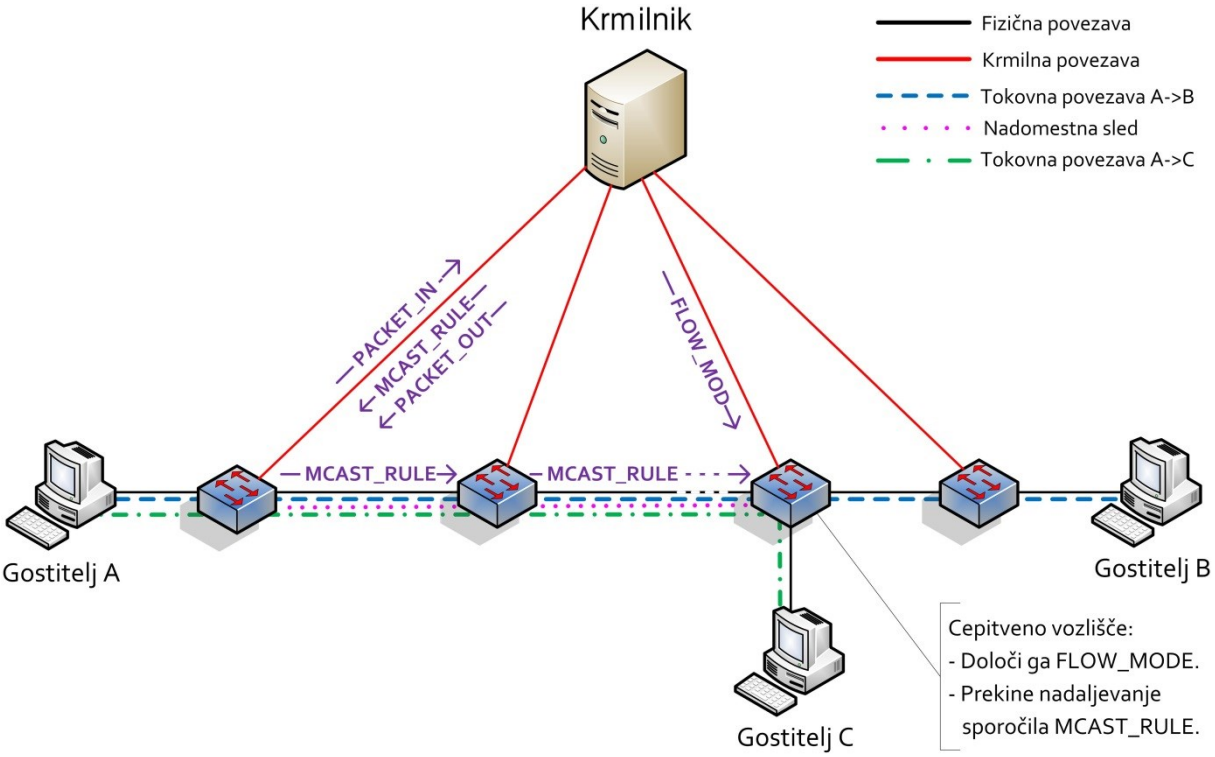
  Stikalo, od katerega se izbrana nadomestna sled ne nadaljuje po isti poti, kot jo zahteva novi podatkovni tok, imenujemo cepitveno vozlišče (ang. *split node*).

4.3 Opis nove metode

Razliko med metodama krmiljenja, predstavljenima v poglavju 3, je možno doseči samo z dodelavo omrežne aplikacije na krmilniku. To pomeni, da za njuno izvedbo lahko uporabimo standardni protokol na južnem vmesniku (npr. OF) in stikala, ki podpirajo ta protokol. Za izvedbo metode z razpršenim reakcijskim krmiljenjem (Metoda III) je potrebna tako nadgradnja omrežne aplikacije na krmilniku kot tudi dodelava protokola in stikal (poglavje 4.5). Te dodelave omogočajo, da je za vzpostavljanje posamezne tokovne povezave za vsa stikala podatkovne ravnine dovolj praktično eno samo krmilno sporočilo *MCAST_RULE*. To sporočilo krmilnik generira in pošlje samo enkrat, zato zakasnitev ni odvisna od števila elementov na poti, vendar je potrebno zagotoviti, da ga prejmejo vsa relevantna stikala.

4.3.1 Predstavitev na primeru

Slika 11 prikazuje primer vzpostavitve tokovne povezave med gostiteljem A in gostiteljem C po Metodi III. Prikazani primer predvideva, da je predhodno vzpostavljena tokovna povezava med gostiteljem A in gostiteljem B. Ta je lahko vzpostavljena z eno od prej opisanih metod ali pa s tehniko proaktivnega krmiljenja (poglavje 2.5.1).



Slika 11: Razpršeno reakcijsko krmiljenje (Metoda III)

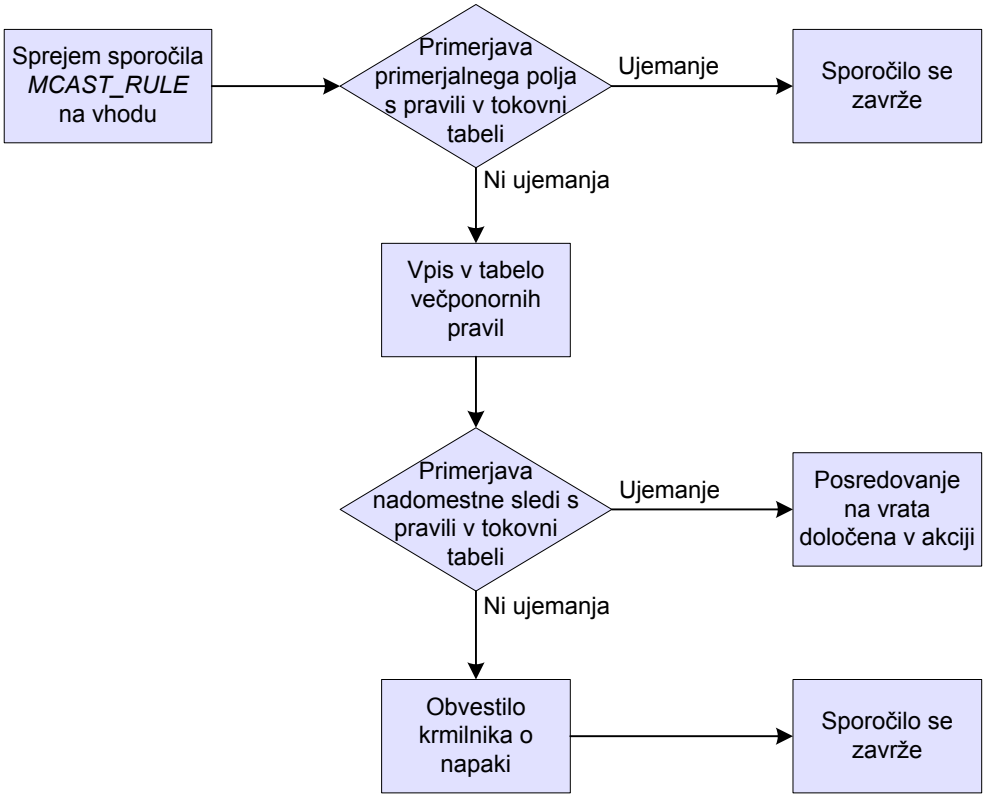
Ko pride prvi paket novega podatkovnega toka na prvo stikalo na poti, ga to pošlje na krmilnik. Krmilnik paket obdela in na prvo stikalo pošlje krmilno sporočilo *MCAST_RULE*. Sporočilo vsebuje podatek o izbrani nadomestni sledi. Stikalo podatek shrani v tabelo večponornih pravil. Nato stikalo po izbrani nadomestni sledi, ki je sicer namenjena posredovanju podatkovnega toka, posreduje tudi sporočilo *MCAST_RULE* naslednjemu stikalu. Stikalo, ki to sporočilo prejme od svojega soseda, ga obravnava na enak način, kot da bi ga prejelo od krmilnika, zato ga vpiše v tabelo in posreduje naprej po sledi. Verigo prekine krmilnik, tako da pošlje dodatno (standardno) sporočilo za vzpostavljanje tokovne povezave *FLOW_MOD*[8] na stikalo, ki predstavlja cepitveno vozlišče. Cepitveno vozlišče prekine nadaljevanje sporočila *MCAST_RULE*. To stori na podlagi prisotnosti pravila za nov tok v standardni tokovni tabeli. Zato vsa stikala po prejemu sporočila *MCAST_RULE* najprej izvedejo primerjavo primerjalnega polja v pravilu s pravili v tokovni tabeli. Če pravilo obstaja, stikalo sporočilo *MCAST_RULE* zavrže brez vpisovanja v tabelo večponornih pravil (Slika 12). Neodvisno od dogajanja na podatkovni ravnini krmilnik kot zadnje (tretje) sporočilo pošlje še *PACKET_OUT* nazaj prvemu stikalu. Ko prvi paket novega podatkovnega toka zapusti cepitveno vozlišče, je z malo sreče že prispel do cilja (primer na Sliki 11), če pa

[8] V praksi je najbolj varno, da krmilnik to sporočilo pošlje kot prvo – pred sporočilom *MCAST_RULE*.

je naslednja točka samo stikalo na poti toka, se celotni postopek krmiljenja po nadomestni sledi lahko ponovi na naslednjem delu omrežja.

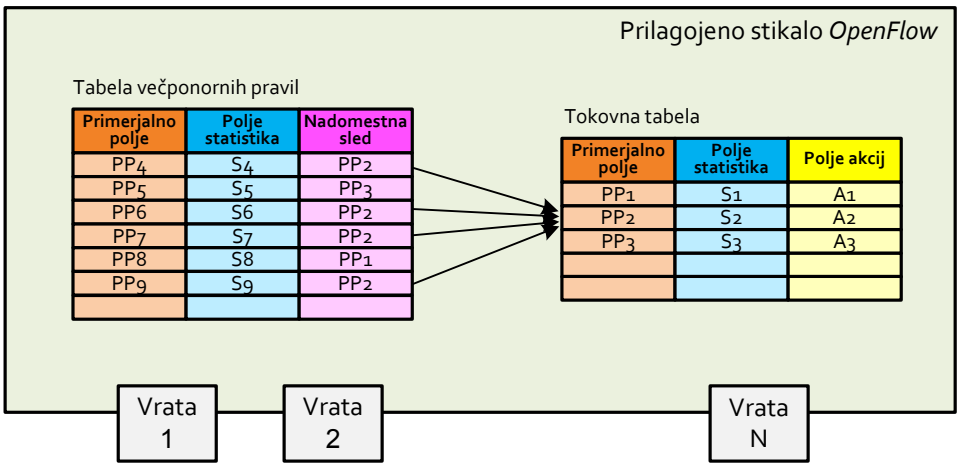
4.3.2 Arhitektura in delovanje stikala

Slika 12 prikazuje obravnavo sporočila *MCAST_RULE* na stikalu, modificiranem za potrebe izvajanja Metode III.



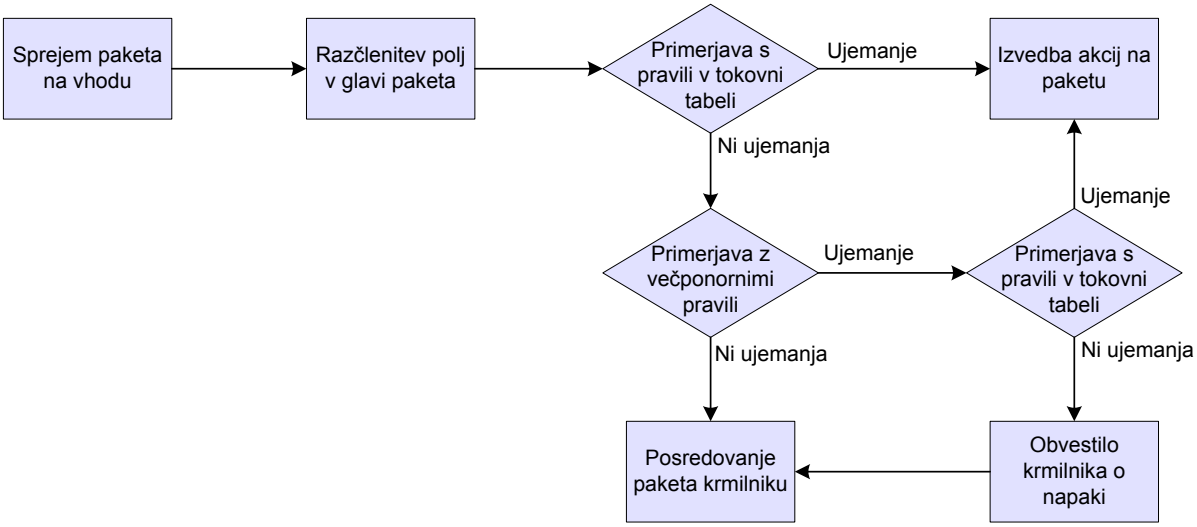
Slika 12: Obravnava sporočila MCAST_RULE na stikalu

Vsak vhodni podatkovni paket se na stikalu, ki podpira opisano metodo krmiljenja, obdela nekoliko drugače kot v primeru standardne podatkovne ravnine (poglavje 2.3). Prilagojeno stikalo OF vsebuje vsaj dve tabeli: standardno tokovno tabelo in tabelo večponornih pravil (Slika 13). Po sprejetju paketa in razčlenitvi polj njegove glave stikalo najprej izvede primerjavo s pravili v tokovni tabeli. V primeru ujemanja se za paket izvede predpisana akcija, v nasprotnem primeru se primerjava izvede še z vpisi v tabeli večponornih pravil. Če ujemanja tudi tu ni, se paket pošlje na krmilnik. V kolikor pride do ujemanja, se iz tabele preberejo podatki o nadomestni sledi. Ti podatki so kar celotno primerjalno polje obstoječega vnosa v tokovni tabeli. Nova primerjava s tokovno tabelo tako praviloma vedno uspe in stikalo lahko izvede akcijo nadomestne sledi. V primeru, da zadnja primerjava vseeno ne uspe, se paket posreduje krmilniku skupaj s sporočilom o napaki. Slika 14 prikazuje potek obravnave podatkovnih paketov znotraj prilagojenega stikala OF.



Slika 13: Tabele prilagojenega stikala OpenFlow

Na Sliki 13 je razviden glavni namen predlagane metode. Vsa tokovna pravila, ki zahtevajo enako akcijo, vsebujejo kazalec na isto vrstico tokovne tabele. Ta kazalec je kar vrednost primerjalnega polja nadomestne sledi. Vidimo, da so posledično tokovne tabele precej krajše – vsebujejo npr. toliko vpisov, kolikor ima stikalo vrat. Dolžina tabele večponornih sporočil je primerljive dolžine, kot je dolžina tokovne tabele pri standardni izvedbi.



Slika 14: Obravnava podatkovnih paketov v prilagojenem stikalu OpenFlow

4.3.3 Izračun karakteristik po primeru

Pri izračunih karakteristik Metode III so v nadaljevanju upoštevane predpostavke:

- Predhodne vzpostavljanje tokovne povezave, ki se uporablja kot nadomestna sled, ni potrebno vključiti v izračune, saj se ta lahko vzpostavi proaktivno ob inicializaciji krmilnika (poglavje 2.5.1).

- Za vzpostavitev nove tokovne povezave je potrebna uporaba samo ene nadomestne sledi, saj uporaba več zaporednih nadomestnih sledi ni nujno potrebna. Tudi to je

možno zagotoviti z ustrezno postavitvijo začetnih, proaktivno vzpostavljenih tokovnih povezav.

- Tudi v primeru več zaporednih nadomestnih sledi ne pride do bistvene spremembe izračuna, saj krmilnik vse naslednje nadomestne sledi lahko vzpostavi nemudoma po vzpostavitvi prve. Novo sporočilo *PACKET_IN* se v tem primeru ne pošilja, zato se tudi časa T_{PACKET_IN} in T_C ne množita.

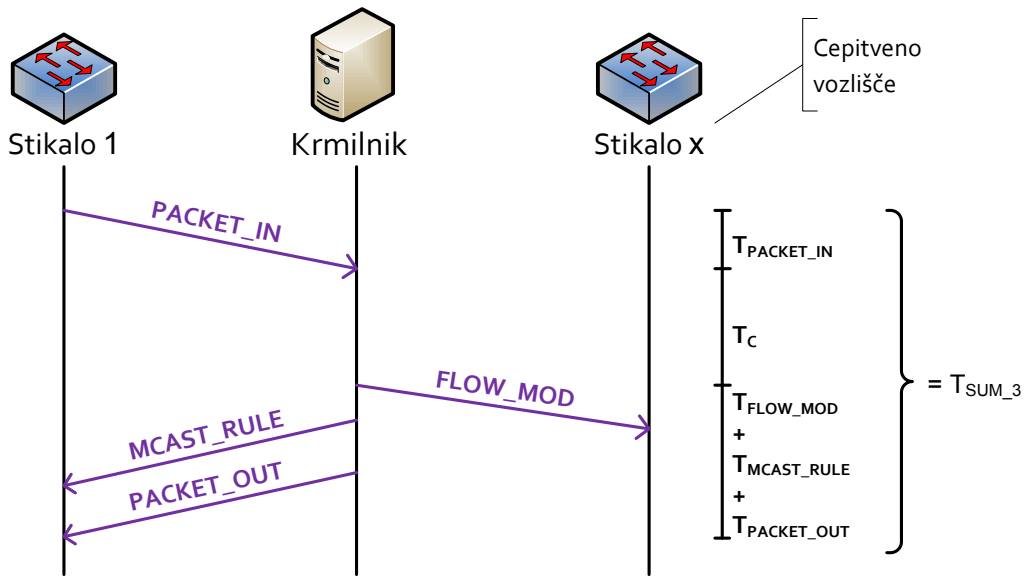
Izračun števila vseh prenosov sporočil na relaciji stikalo–krmilnik za poljubno dolžino poti po Metodo III je zelo enostaven:

$$M_{SUM\_3} = 4 \qquad (3.0)$$

Razlika v številu sporočil med Metodo II in Metodo III je tako:

$$\Delta M_{23} = M_{SUM\_2} - M_{SUM\_3} = N - 2 \qquad (3.1)$$

Slika 15 prikazuje časovni potek razpršenega reakcijskega vzpostavljanja tokovne povezave.



Slika 15: Časovni potek razpršenega reakcijskega krmiljenja (Metoda III)

Skupno zakasnitev prvega paketa novega podatkovnega toka za poljubno dolžino poti, izraženo s časom obdelave na krmilniku (T_C)[9] ter časom, potrebnim za prenos posameznega sporočila (T_M), za Metodo III lahko izračunamo po enačbi:

$$T_{SUM\_3} = T_C + M_{SUM\_3} \times T_M = T_C + 4 \times T_M \qquad (3.2)$$

Z Metodo III se zakasnitev zmanjša glede na Metodo II za:

$$\Delta T_{23} = T_{SUM\_2} - T_{SUM\_3} = T_C + (N + 2) \times T_M - T_C - 4 \times T_M = (N - 2) \times T_M \qquad (3.3)$$

[9] Tako kot v poglavju 3.2 tudi v tem primeru zanemarimo dodatni čas obdelave na krmilniku.

Zaključimo lahko, da predlagana metoda prinaša precejšen prihranek pri številu potrebnih krmilnih sporočil – namesto linearne odvisnosti dobimo konstantno odvisnost[10], kar velja tudi za čas, potreben za vzpostavitev nove tokovne povezave.

4.4 Hipotetično omrežje

Predpostavimo hipotetično omrežje SDN, ki npr. obsega 500 stikal in pri katerem je povprečna dolžina tokovne povezave dolga npr. 20 stikal. Denimo, da je na vsako stikalo v povprečju priključenih 10 gostiteljev; vseh gostiteljev skupaj je torej 5.000. Če imamo granulacijo podatkovnega prometa po podatkovnih tokovih določeno do glave TCP, vsak gostitelj ustvari toliko podatkovnih tokov, kolikor ima aktivnih omrežnih aplikacij. Predpostavimo, da so vsi gostitelji običajni domači uporabniki, ki imajo istočasno aktivnih med 10 in 100 podatkovnih tokov in ustvarijo v povprečju en nov podatkovni tok[11] na sekundo.

Krmilnik opisanega hipotetičnega omrežja mora v povprečju vsako sekundo ustvariti 5.000 novih tokovnih povezav[12] na 20 stikalih. Skupaj torej krmilnik izvede 100.000 vpisov na sekundo. Z uporabo enačb, izpeljanih v poglavju 4.3.3, lahko za opisano omrežje izračunamo doprinos Metode III v primerjavi z Metodo I in Metodo II po obravnavanih karakteristikah:

Število vseh izmenjanih sporočil med krmilnikom in stikali za Metodo I izračunamo po enačbi (1.0):

$M_{SUM\_1} = 3 \times N = 3 \times 20 = 60$ sporočil/tok $\Rightarrow$ $60 \times 5.000 = 300.000$ sporočil/s

Število vseh izmenjanih sporočil med krmilnikom in stikali za Metodo II izračunamo po enačbi (2.0):

$M_{SUM\_2} = N + 2 = 22$ sporočil/tok $\Rightarrow$ $22 \times 5.000 = 110.000$ sporočil/s

Število vseh izmenjanih sporočil med krmilnikom in stikali za Metodo III izračunamo po enačbi (3.0):

$M_{SUM\_3} = 4$ sporočila/tok $\Rightarrow$ $4 \times 5.000 = 20.000$ sporočil/s

Po enačbi (3.1) izračunamo še razliko v številu sporočil med Metodo II in Metodo III:

$\Delta M_{23} = N - 2 = 18$ sporočil/tok $\Rightarrow$ $18 \times 5.000 = 90.000$ sporočil/s

Za zakasnitev prvega paketa novega podatkovnega toka izračunamo le razliko med Metodo II in Metodo III po enačbi (3.3):

$\Delta T_{23} = (N - 2) \times T_M = 18 \times T_M$

Iskana razlika v zakasnitvah je torej 18 časov prenosa krmilnega sporočila. Zanima pa nas tudi razmerje med obema zakasnitvama:

$T_3/T_2 = (T_C + 4 \times T_M)/(T_C + (N + 2) \times T_M)$

[10] To velja tudi, če bi bilo potrebno vzpostaviti več nadomestnih sledi zaporedno.
[11] Nov podatkovni tok povzroči npr. vsaka nova HTTP-povezava, sprememba kanala na IP-televiziji ipd.
[12] Te povezave se po izteku časovnika samodejno brišejo.

Razmerja ni mogoče izračunati, če ne določimo razmerja med časom obdelave sporočila na krmilniku in časom prenosa krmilmilnega sporočila. To razmerje je sicer zelo odvisno od konkretnega omrežja, za naše hipotetično omrežje lahko predpostavimo:

$T_C \approx T_M \approx T$

Potemtakem je razmerje zakasnitev:

$T_3/T_2 = (5 \text{ x } T)/((N + 3) \text{ x } T) = 5T/23T = 0{,}22$

Vidimo, da je predlagana metoda z razpršenim reakcijskim krmiljenjem na velikih omrežjih zelo učinkovita. Za naše hipotetično omrežje smo izračunali, da s to metodo krmilnik vsako sekundo razbremenimo za pošiljanje 90.000 sporočil v primerjavi z Metodo II, kar predstavlja kar 82 % razbremenitev. Povprečna zakasnitev pri vzpostavljanju tokovne povezave pa je pri Metodi III za 78 odstotkov manjša kot v primeru krmiljenja po Metodi II.

Pri zgornjih izračunih se predpostavlja, da je vzpostavljanje nove tokovne povezave po Metodi III vedno izvedeno z uporabo ene same nadomestne sledi. To je tudi v praksi možno zagotoviti, saj v času inicializacije omrežja krmilnik lahko proaktivno vpiše nekaj glavnih tokovnih povezav preko celotnega omrežja. Če so te tokovne povezave postavljene dovolj dobro, je privzeta predpostavka tudi izpolnjena.

V kolikor bi vseeno prišlo do potrebe po uporabi več nadomestnih sledi na tokovno povezavo, se karakteristike ne spremenijo bistveno. Če bi predpostavili, da za vsako povezavo potrebujemo tri nadomestne sledi, bi po zgornjih izračunih dobili rezultata: za število zmanjšanja vseh sporočil bi namesto 90.000 dobili 50.000, pri zakasnitvi bi namesto 78-odstotne dobili 48-odstotno izboljšavo. Tudi v tem primeru je razlika z Metodo II še vedno občutna in kaže na veliko učinkovitost Metode III.

Na izbranem hipotetičnem omrežju smo tako teoretično dokazali, da predlagana metoda z razpršenim reakcijskim krmiljenjem izpolnjuje zastavljene cilje po razbremenitvi krmilnika ter cilje po zmanjšanju zakasnitve pri vzpostavljanju nove tokovne povezave. To pomeni, da bo z večanjem omrežja do morebitne preobremenitve prišlo bistveno kasneje ter da bo odzivnost omrežja bistveno bolj skladna z zahtevami po delovanju v realnem času.

Omeniti velja tudi, da nekaj več proaktivno vpisanih tokovnih povezav zagotovi, da ima krmilnik možnost izbire pri določevanju nadomestne sledi in bo lahko izvajal funkcijo porazdeljevanja obremenitve fizičnih povezav. To omogočimo s tako postavitvijo povezav, da gredo skozi vsako stikalo dve ali tri dvosmerne po različnih poteh[13]. Tudi v primeru izpada določenega dela poti lahko krmilnik pri Metodi III enostavno preusmeri veliko število tokov tako, da z enim samim sporočilom *FLOW_MOD* spremeni akcijo ustreznega pravila v tokovni tabeli. Posledično bodo vsi tokovi, ki uporabljajo to pravilo kot nadomestno sled, preusmerjeni na novo, redundantno sled. Tako je s hitreje izvedenim preklopom z novo metodo izboljšana tudi odpornost omrežja SDN. Vse prednosti predlagane metodo so zbrane v poglavju 4.6.1.

[13] Predpostavlja se, da fizična topologija omrežja to omogoča.

4.5 Prilagoditve, potrebne za izvedbo predlagane metode

Realizacija v poglavju 4.3 predlagane metode zahteva spremenjeno logiko krmilne ravnine, spremembo arhitekture in poteka obdelave paketov na podatkovni ravnini ter formulacijo dodatnega sporočila v protokolu za komunikacijo med obema ravninama. Za primer protokola *OpenFlow* so te modifikacije že nakazane v poglavju 4.3.2. V naslednjih razdelkih so modifikacije povzete, nekateri detajli pa so predstavljeni podrobneje.

4.5.1 Prilagoditev krmilne ravnine

Prilagojena programska oprema krmilnika mora za uspešno izvajanje predlagane metode krmiljenja zagotoviti:

- prepoznavanje možnih in izbiro najustreznejših nadomestnih sledi,
- gradnjo in pošiljanje krmilnih sporočil za uspešno vzpostavljanje tokovne povezave po izbrani nadomestni sledi,
- preprečitev izteka časovnika za tokovno pravilo, ki je uporabljeno kot nadomestna sled,
- preprečitev nenadzorovanega širjenja večponornega sporočila z določitvijo cepitvenega vozlišča preko krmiljenja standardne tokovne tabele,
- v primeru veriženja več zaporednih tokovnih nadomestnih sledi, je dodatno zaželeno, da vse naslednje nadomestne sledi vzpostavi nemudoma po vzpostavitvi prve. Tako ne pride do ponovnega pošiljanja istega podatkovnega paketa na krmilnik.

Pri izpolnjevanju gornjih zahtev mora krmilnik upoštevati:

- da ne proži procedure vzpostavljanja nadomestne sledi, če je prvo stikalo hkrati tudi zadnje na poti, torej sta izvor in cilj priključena na različna vrata istega stikala,
- da ne proži procedure vzpostavljanja nadomestne sledi, če je ugotovljeno, da je naslednje stikalo na poti cepitveno vozlišče, saj takšno vzpostavljanje nadomestne sledi ne bi prineslo prednosti,
- da stikalo na poti tokovne povezave, ki paketom spremeni katero od polj v glavi in je to polje vključeno tudi v primerjavo s tokovnim pravilom, obravnava enako kot cepitveno vozlišče. Če se namreč spremeni polje za primerjavo, vsebina večponornega krmilnega sporočila ni več aktualna.

4.5.2 Prilagoditve podatkovne ravnine

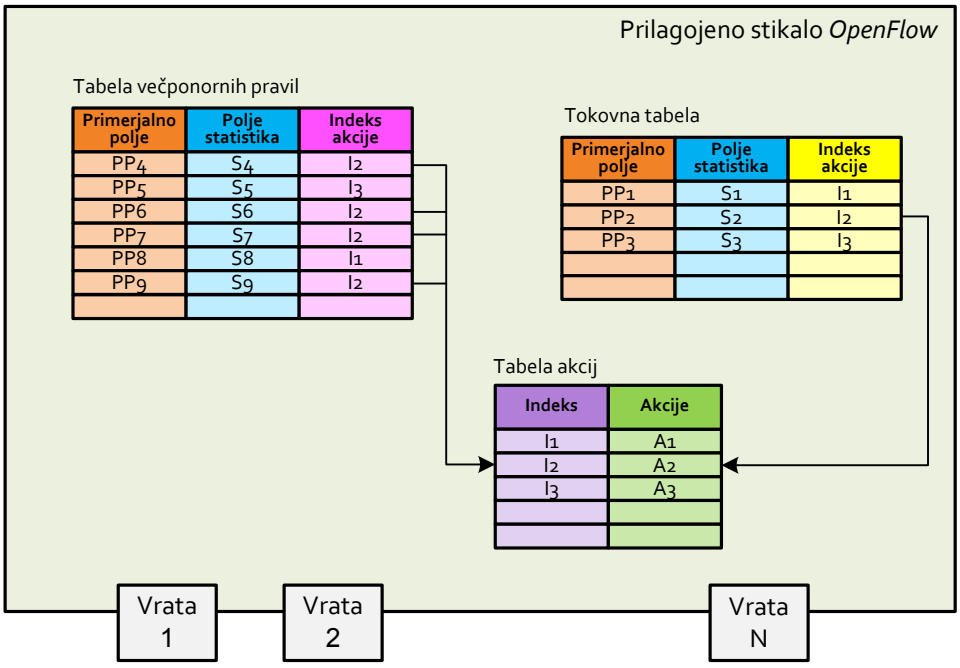
Prilagojena programska in strojna oprema stikal mora za uspešno izvajanje predlagane metode krmiljenja zagotoviti:

- sprejem, prepoznavanje in ustrezno obravnavanje večponornega krmilnega sporočila v skladu z diagramom poteka na Sliki 12,
- obstoj ustrezne tabele za večponorna sporočila (Slika 13),

- spremenjeno obravnavo podatkovnih paketov v skladu z diagramom poteka na Sliki 14, pri čemer je posebej pomembna prioriteta tabel – v primeru ujemanja paketa s pravili v obeh tabelah, ima vedno prednost standardna tokovna tabela.

Dodatna izboljšava podatkovne ravnine

Na diagramu poteka posredovanja podatkovnih paketov na Sliki 14 vidimo, da so v večini primerov potrebne tri primerjave s tabelama na Sliki 13. To je neugodno predvsem za programske izvedbe stikal, saj vsaka primerjava prinese določeno zakasnitev pri posredovanju paketov. Na Sliki 16 je prikazana modificirana arhitektura tabel v stikalu glede na Sliko 13. Modifikacija omogoča, da je posredovanje podatkovnih paketov optimalnejše.



Slika 16: Optimalnejša arhitektura tabel v stikalu

Arhitektura na Sliki 16 vsebuje dodatno indeksno tabelo akcij. Indeksi zagotovijo direktni dostop do vrstice v tabeli, tako iz tabele tokovnih pravil kakor tudi iz tabele večponornih pravil. Pri posredovanju podatkovnih paketov ni potrebno še drugo pregledovanje tokovne tabele (Slika 14), ki je namenjeno iskanju ustrezne akcije, zato je zakasnitev pri posredovanju paketov manjša.

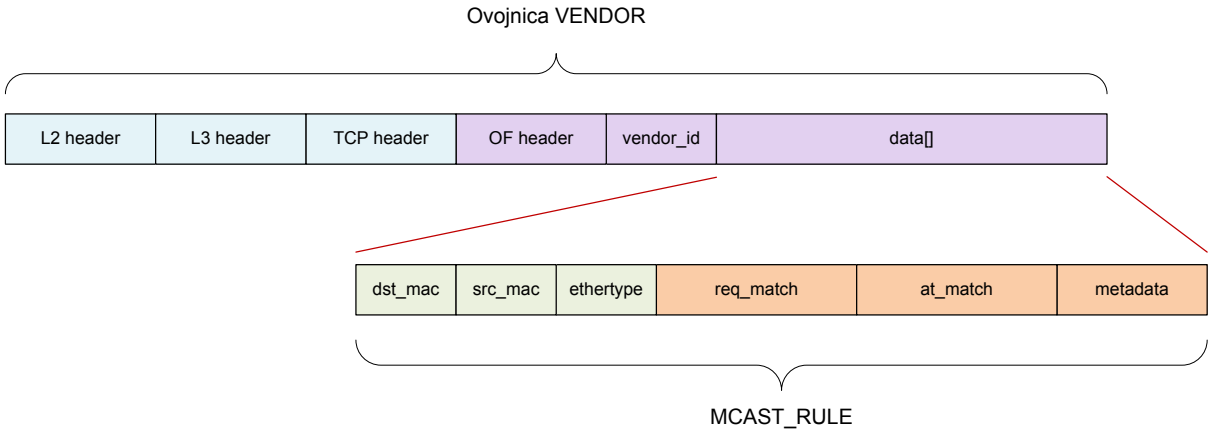
Modifikacija zahteva nekoliko spremenjeno obravnavo sporočila MCAST_RULE. Pri vpisovanju pravila v večponorno tabelo je sedaj potrebno vpisati indeks akcije in ne primerjalno polje nadomestne sledi. To lahko dosežemo z manjšo spremembo vrstnega reda pri obdelavi sporočila (Slika 12) – primerjavo nadomestne sledi s pravili v tokovni tabeli izvedemo pred vpisom vrstice v tabelo večponornih sporočil. V primeru ujemanja preberemo indeks akcije iz tokovne tabele, ki ga uporabimo pri vpisu vrstice v tabelo večponornih pravil.

Kot rečeno, je modifikacija koristna za programske izvedbe stikal, ki se po večini uporabljajo v simulacijskih okoljih, kot je npr. okolje, predstavljeno v poglavju 5. Podobno koristna je tudi za strojne izvedbe, ki za primerjalne tabele ne uporabljajo pomnilnika CAM. V obeh primerih lahko s takšno izvedbo zmanjšamo zakasnitev pri posredovanju paketov. V primeru

stroje izvedbe s pomnilnikom CAM z uporabo modifikacije lahko zmanjšamo porabo dragega pomnilnika CAM, saj je indeks bistveno manjši podatek kot akcija ali podatek o nadomestni sledi. Indeksna tabela akcij je v vsakem primeru implementirana v običajnem, veliko cenejšem pomnilniku RAM (ang. *Random-access Memory*).

4.5.3 Prilagoditev protokola

Za izvedbo predlagane metode z razpršenim oddajanjem krmilnih sporočil je v protokolu za krmiljenje podatkovne ravnine (npr. OF) potrebno definirati dodatno, že večkrat omenjeno večponorno krmilno sporočilo *MCAST_RULE*. Sporočilo je tipa krmilnik proti stikalu (poglavje 2.6.2) in je po svoji vlogi podobno sporočilu *FLOW_MOD*. Glavna razlika je v tem, da sporočilo *MCAST_RULE* doseže in s tem krmili več stikal, sporočilo *FLOW_MOD* pa samo enega. Namen in uporaba sporočila je podrobno predstavljena v poglavju 4.3, Slika 17 prikazuje še format sporočila.



Slika 17: Format sporočila MCAST_RULE

Za namene prototipnega testiranja je sporočilo *MCAST_RULE* ovito v sporočilo *VENDOR*, ki je po standardu predvideno za tovrstne potrebe (poglavje 2.6.2). Ovojnico sestavljajo polja *OF header, vendor_id* in *data[]*:

- *OF header* – polje glava (64 bitov), ki je enako za vsa sporočila OF. Znotraj glave so polja: verzija, tip, dolžina in identifikator transakcije (poglavje 2.6.2).
- *vendor_id* – identifikator sporočila (32 bitov). Polje se uporablja za identifikacijo različnih eksperimentalnih sporočil.
- *data[]* – polje podatki. Vsebuje vsebino eksperimentalnega sporočila poljubne dolžine.

Ovojnica *VENDOR* poleg polj protokola OF vsebuje tudi običajne glave drugega, tretjega in četrtega (transportnega) nivoja (*L2 header / L3 header / TCP header*), ki so potrebne za prenos sporočila od krmilnika do stikala. Te glave vsebujejo ciljne naslove za enega prejemnika (ang. *unicast*). V našem primeru je to naslov stikala, na katerem se nadomestna sled začne. Za sporočilo *MCAST_RULE* pa pričakujemo, da dospe na vsa stikala na tej poti. Znotraj polja *data[]* imamo zato sporočilo ovito v nov okvir drugega nivoja – L2 (ang.

Level 2) oz. okvir Ethernet, ki kot ciljni naslov vsebuje kodo oz. naslov za razpršeno oddajanje ter posebno identifikacijo v polju *ethertype*. Takšno sporočilo lahko nespremenjeno dospe na vsa relevantna stikala. Seveda mora prvo stikalo odstraniti celotno ovojnico *VENDOR* in posredovati le okvir Ethernet znotraj polja *data[]*.

Polje *data[]* vsebuje jedro sporočila *MCAST_RULE* (Slika 17), ki ga sestavljajo:

- glava Ethernet:
  - *dst_mac* – ciljni naslov MAC (ang. *Media Access Control*) (48 bitov). Vsebuje naslov za razpršeno pošiljanje (npr. 01-80-C2-00-00-FF[14]).
  - *src_mac* – izvorni naslov MAC (48 bitov). Vsebuje izvorni naslov krmilnika.
  - *ethertype* – tip Etherneta (16 bitov). Vsebuje posebno identifikacijsko številko, ki označuje sporočilo *MCAST_RULE* (npr. 0xFFEF[15]).
- podatki *MCAST_RULE*:
  - *req_match* – želeni tok (320 bitov). Vsebuje primerjalno polje želenega podatkovnega toka.
  - *at_match* – nadomestna sled (320 bitov). Vsebuje primerjalno polje podatkovnega toka, ki je v delu ali celoti izbran za nadomestno sled.
  - *metadata* – metapodatki (192 bitov). Vsebina je odvisna od implementacije. Vsebuje lahko ekvivalentna polja kot sporočilo *FLOW_MOD* (brez polj *header*, *match* in *action[]*) (poglavje 2.6.2) ali pa samo indekse tabel, kot je to v primeru implementacije, opisane v poglavju 5.3.

4.6 Prednosti in slabosti predlagane metode

Predstavitev nove metode zaključujemo s povzetkom vseh prednosti, ki jih ta prinaša, opozarjamo pa tudi na njene pomanjkljivosti.

4.6.1 Prednosti razpršenega reakcijskega krmiljenja

Že večkrat omenjene prednosti krmiljenja z razpršenim reakcijskem krmiljenjem lahko strnemo v naslednjih točkah:

- Manjša obremenjenost krmilnika.

  Krmilnik ustvari in odda manjše število krmilnih sporočil in s tem prihrani pri procesorski moči. Krmilnik enake zmogljivosti tako lahko po predlagani metodi krmili precej večje omrežje.

[14] Naslov s strani organizacije za standardizacijo IEEE (ang. Institute of Electrical and Electronics Engineers) še ni dodeljen [38].
[15] Koda je s strani organizacije za standardizacijo IEEE opredeljena kot eksperimentalna [39].

- Manj problematična razširljivost.

  V poglavju 4.3.3 je bilo pokazano, da se z daljšanjem tokovne povezave obremenjenost krmilnika le malenkostno povečuje. Pri povečevanju omrežja tako le robna stikala prispevajo k povečevanju zahtevnosti po procesorski moči na krmilniku. Potreba po vključevanju dodatnih krmilnikov ob povečevanju omrežja je tako bistveno manjša.

- Manjša zakasnitev pri vzpostavljanju tokovne povezave.

  Zaradi manjšega števila izmenjanih sporočil med krmilnikom in stikali prvi paket podatkovnega toka napreduje hitreje po omrežju.

- Boljša odpornost omrežja

  V primeru izpada določene povezave lahko krmilnik z enim samim krmilnim sporočilom spremeni akcijo, povezano z izpadlo povezavo. Nova akcija tako preusmeri vse tokove, ki to akcijo uporabljajo kot nadomestno sled.

4.6.2 Pomanjkljivosti razpršenega reakcijskega krmiljenja

- Zakasnitev pri posredovanju paketov na stikalih.

  Zaradi primerjave z dodatno primerjalno tabelo se za vsak podatkovni paket nekoliko podaljša čas prehoda skozi stikalo. Za stikala s pomnilnikom CAM je ta čas zanemarljivo majhen, pri ostalih se vsaka primerjava s tabelo, ki ima veliko vnosov, pozna na času prehoda. Olajševalna okoliščina je sicer, da predlagana metoda močno zmanjša število vnosov v originalni tokovni tabeli, tako da se s tem nekoliko kompenzira dodatni čas zaradi dodatne primerjave. Druga olajševalna okoliščina je, da v stikalih, ki uporabljajo cevovod več tokovnih tabel (poglavje 2.6.1), ena dodatna primerjava prinese relativno majhen pribitek. Najbolj se tako dodatna primerjava odrazi na povečanju zakasnitve pri posredovanju paketov v primeru programsko izvedenih stikal, ki v originalni izvedbi vsebujejo samo eno tokovno tabelo.

- Robustnost rešitve je v realnih okoljih lahko nezadostna.

  Če bi se katero od sporočil *MCAST_RULE* zaradi kakršnekoli napake razširilo izven predvidenega območja, bi prišlo do napačnega krmiljenja stikal in posledično do odpovedi omrežja. Področje robustnosti je zato v prihodnje potrebno še raziskati in predlagati ustrezne zaščitne mehanizme.

- Varnost rešitve je v realnih okoljih lahko nezadostna

  Če bi zlonamernemu uporabniku uspelo v omrežje poslati takšno sporočilo *MCAST_RULE*, ki bi povzročalo nekontrolirane vnose v tabele stikal, bi lahko prišlo do napačnega delovanja omrežja. Tudi področje varnosti je zato v prihodnje potrebno še raziskati in predlagati ustrezne zaščitne ukrepe.

5 POSTAVITEV OKOLJA ZA SIMULACIJO

V okviru magistrskega dela je bilo postavljeno simulacijsko okolje za preverjanje učinkovitosti nove metode krmiljenja, predstavljene v poglavju 4 (Metoda III). Preverjanje je bilo opravljeno v primerjavi z obema obstoječima metodam krmiljenja, opisanima v poglavju 3 (Metoda I in Metoda II). V nadaljevanju so najprej natančno opredeljeni cilji, preverjanje katerih mora zagotoviti okolje. Nato sledi opis simulacijskega okolja ter razlaga v okviru naloge razvite programske opreme, ki predstavlja implementacijo omenjenih metod krmiljenja znotraj okolja. Nazadnje so zapisana še navodila za zagon simulacij.

Rezultati simulacije, ki jih imenujemo tudi opravljene meritve na simulacijskem okolju, so predstavljeni v poglavju 6.

5.1 Cilji simulacije

Cilji simulacijskega okolja z implementirano metodo krmiljenja z razpršenim pošiljanjem krmilnih sporočil so usmerjeni k zagotavljanju izvedbe naslednjih testov:

- Dokaz koncepta – PoC (ang. *Proof of Concept*). S simulacijo želimo v prvi vrsti dokazati, da predlagana metoda krmiljenja ohranja delovanje omrežja.
- Preveriti zmanjšanje števila sporočil, potrebnih za vzpostavitev tokovne povezave v primerjavi z obstoječima metodama krmiljenja.
- Izmeriti izboljšanje časov pri vzpostavljanju tokovne povezave v primerjavi z obstoječima metodama krmiljenja.
- Preveriti morebitno povečanje zakasnitve pri prehodu podatkovnih paketov preko stikal.
- Zmanjšana obremenitev krmilnika v primerjavi z obstoječima metodama krmiljenja je bila prikazana teoretično (poglavje 4.3.3). Zaradi zahtevnosti po bistveno večjem simulacijskem omrežju kot v prejšnjih točkah praktični preizkus te točke ostaja izziv za bodoče delo.

5.2 Opis okolja

Okolje za praktičen del je bilo pripravljeno na običajnem računalniku, na katerem je v navideznem okolju potekala simulacija omrežja SDN. Navidezni računalnik z operacijskim sistemom *Linux* različice *Ubuntu* [44] je bil s pomočjo orodja *Vagrant* [46] postavljen na platformi za navidezne računalnike *VirtualBox* [45]. Na postavljenem navideznem računalniku je s programom *Mininet* [47] možno relativno enostavno simulirati oziroma natančneje – emulirati, omrežja različnih topologij.

5.2.1 Priprava navideznega okolja

Postavitev navideznega računalnika je s pomočjo orodja *Vagrant* izredno hitra in nezahtevna. To orodje omogoča shranjevanje in izmenjavo navideznih računalnikov v obliki slik (ang. *image*) ter enostavno konfiguriranje navideznega računalnika preko datoteke *vagratfile*. S pomočjo te datoteke lahko na enem mestu nastavimo vrsto nastavitev operacijskega sistem navideznega računalnika, ki jih je sicer potrebno po zagonu nastavljati na različnih mestih sistema.

Za postavitev navideznega okolja je tako potrebno na gostiteljski računalnik namestiti ustrezni različici platforme *VirtualBox* in orodja *Vagrant*. Ko program namestimo, moramo pridobiti še ustrezno sliko navideznega računalnika. Veliko slik za različne projekte je na voljo na [48], za potrebe magistrskega dela pa je bila uporabljena slika navideznega računalnika, ki je dostopna na [50] in je bila pripravljena za potrebe spletnega izobraževanja *Coursera SDN* v letu 2015.

Slika navideznega računalnika vsebuje omenjeno datoteko nastavitev *vagratfile*, v kateri je možno parametre prilagoditi lastnim potrebam oz. zmožnostim gostiteljskega računalnika še pred zagonom navideznega računalnika. Nazadnje preko ukazne vrstice vnesemo ukaz *vagrant up* za start navideznega računalnika.

Na navidezni računalnik se najenostavneje priključimo preko terminalskega dostopa SSH (ang. *Secure Shell*). Da se na gostiteljskem računalniku prikažejo tudi morebitna pojavna okna, odprta s strani navideznega računalnika, mora imeti ta zagnan strežnik za okna X. Za oboje, dostop SSH in strežnik oken X, v našem primeru poskrbi program MobaXterm [49].

5.2.2 Orodje *Mininet*

Mininet je orodje, ki na običajno zmogljivem računalniku zelo učinkovito emulira omrežja SDN. Emulirana omrežja lahko vsebujejo tudi več tisoč gostiteljev (ang. *hosts*), ki so med seboj povezani preko poljubne topologije stikal. Stikala pa krmili eden ali več krmilnikov. Omejitve se pokažejo šele ob pošiljanju velike količine podatkovnega prometa preko emuliranega omrežja. Vsota prepustnosti vseh povezav omrežja namreč ne more preseči prepustnosti razpoložljive procesne moči.

Za namestitev programa obstaja več opcij [51]. V našem primeru je bila izbrana priporočena opcija z uporabo slike navideznega računalnika. Izbrana je bila slika, omenjena v prejšnjem razdelku.

Omrežje z orodjem *Mininet* lahko postavimo preko klica že pripravljenih funkcij za posamezne tipe topologij, npr.: zvezdna (*single*), linearna (*linear*), drevesna (*tree*), torus (*torus*) topolgija. Pri klicu je potrebno navesti le parametre topologije, kot so: število stikal, število gostiteljev, globina drevesa ipd. Po želji lahko napišemo čisto svojo funkcijo, ki bo zgradila kakršnokoli topologijo si bomo zamislili [30]. Za programiranje se uporablja programski jezik *Python*.

Mininet vsebuje tudi programček z grafičnim vmesnikom, preko katerega je možno sestaviti omrežje. Imenuje se *MiniEdit*, pregledna navodila za njegovo uporabo so na voljo na [52]. To je na prvi pogled privlačna možnost, vendar ni primerna za postavljanje velikih omrežij.

Za označevanje elementov omrežja *Mininet* uporablja oznake: c_i za krmilnike ($i = 0..n$), s_i za stikala ($i = 1..n$) in h_i za gostitelje ($i = 1..n$). Oznake gostiteljev se spremenijo, če imajo omrežje z več stikali in na vsakem stikalu več kot enega gostitelja. V tem primeru so gostitelji označeni z oznako $h_i s_j$ ($i = 1..n$[16], *j = številka stikala, na katero je povezan gostitelj*). Te oznake so privzete za vse vgrajene funkcije. V kolikor napišemo program za lastno topologijo, so oznake poljubne.

Povezave med elementi so vedno predstavljene z oznako para elementov, ki jih povezuje npr. (*h1, s1*). Vmesniki na elementih so označeni z oznako elementa in oznako vrat (eth_i), kot npr. *s1-eth1*. Povezave do krmilnika so vse vezane na vmesnik lokalne povratne zanke *lo* (ang. *loopback*). Če uporabimo orodje za zajemanje prometa *Wireshark* [54], lahko z izbiro ustrezne oznake zajemamo promet na kateremkoli vmesniku kateregakoli elementa v omrežju.

Ko zaženemo program *Mininet*, se najprej zgradi omrežje, določeno s parametri v ukazni vrstici za zagon (glej primer spodaj). Za parametre, ki niso navedeni, se uporablja privzete vrednosti. Privzeta vrednost za krmilnik je POX (poglavje 5.2.3), za stikalo *OpenFlow kernel switch* [55] in za protokol *OpenFlow* (poglavje 2.6). Ko je omrežje zgrajeno, dobimo *Mininet* vrstični ukazni vmesnik oz. CLI (ang. *Command Line Interface*) za upravljanje celotnega omrežja. Tu lahko prožimo ukaze za posamezne elemente omrežja, ki lahko predstavljajo akcije na ali med elementi. Na primer ukaz »*h1 ping h2«* pomeni, »na gostitelju *h1* izvedi ukaz: *ping h2*«. Primer še močnejšega ukaza je »*pingall«*, ki sproži preverjanje povezljivosti med vsemi gostitelji (vsak z vsakim po en *ping* prožen z obeh strani).

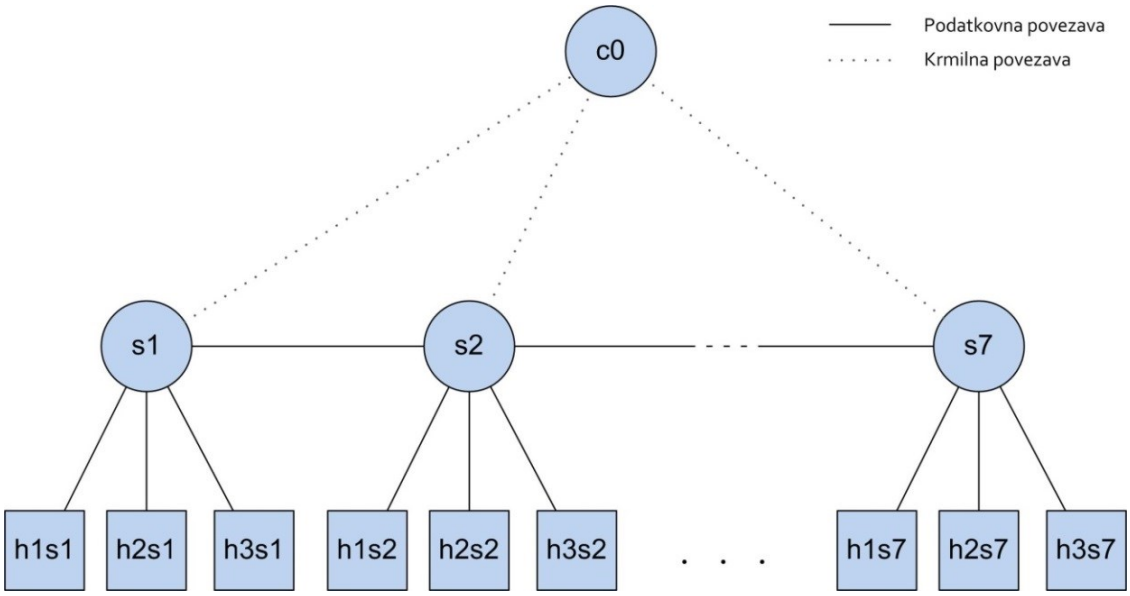
V nadaljevanju je na primeru prikazana osnovna uporaba orodja *Mininet*, podrobnejša navodila za uporabo so na voljo na [53].

Primer postavitve omrežja

Za primer vzemimo linearno omrežje s sedmimi stikali, tremi gostitelji na vsakem stikalu in enim krmilnikom (Slika 18). Omrežje zgradimo z ukazom:

```
$ sudo mn --topo linear,7,3
```

[16] V tem primeru n ni število vseh gostiteljev, ampak število gostiteljev, povezanih na posamezno stikalo.



Slika 18: Primer omrežja Mininet

Ko program zgradi omrežje, dobimo ukazno vrstico *mininet>*. Tu lahko npr. preizkusimo ukaz *ping* preko celotne dolžine omrežja (v povezavo so vključena vsa stikala topologije):

mininet> h1s1 ping h1s7

Rezultat prikazuje Slika 19.

```
mininet> h1s1 ping h1s7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=1 ttl=64 time=30.2 ms
64 bytes from 10.0.0.7: icmp_seq=2 ttl=64 time=0.611 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=64 time=0.106 ms
64 bytes from 10.0.0.7: icmp_seq=4 ttl=64 time=0.099 ms
64 bytes from 10.0.0.7: icmp_seq=5 ttl=64 time=0.085 ms
64 bytes from 10.0.0.7: icmp_seq=6 ttl=64 time=0.076 ms
^C
--- 10.0.0.7 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5000ms
rtt min/avg/max/mdev = 0.076/5.212/30.297/11.219 ms
mininet>
```

Slika 19: Ping preko celotnega omrežja (prvič)

Na Sliki 19 vidimo, da ima prvi paket v sekvenci bistveno daljši čas odgovora kot preostali. Del tega časa pripada protokolu ARP (ang. *Address Resolution Protocol*), del pa pripada protokolu *OpenFlow* za vzpostavljanje dveh tokovnih povezav (vsake v eno smer).

Z ukazom *dpctl del-flows* izbrišemo vse obstoječe tokovne povezave, medtem ko tabele ARP na gostitelju ostanejo popolnjene[17]. Novo proženje ukaza *ping* tako pri prvem paketu pokaže le zakasnitev zaradi vzpostavljanja tokovne povezave v našem omrežju (Slika 20).

[17] Enako stanje bi dosegli z opcijo "--arp" pri ukazu za gradnjo omrežja. Opcija pomeni statično popolnitev tabel ARP.

```
mininet> h1s1 ping h1s7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=1 ttl=64 time=16.8 ms
64 bytes from 10.0.0.7: icmp_seq=2 ttl=64 time=0.424 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=64 time=0.135 ms
64 bytes from 10.0.0.7: icmp_seq=4 ttl=64 time=0.079 ms
64 bytes from 10.0.0.7: icmp_seq=5 ttl=64 time=0.084 ms
64 bytes from 10.0.0.7: icmp_seq=6 ttl=64 time=0.076 ms
64 bytes from 10.0.0.7: icmp_seq=7 ttl=64 time=0.080 ms
64 bytes from 10.0.0.7: icmp_seq=8 ttl=64 time=0.080 ms
64 bytes from 10.0.0.7: icmp_seq=9 ttl=64 time=0.081 ms
^C
--- 10.0.0.7 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8001ms
rtt min/avg/max/mdev = 0.076/1.985/16.834/5.251 ms
mininet>
```

Slika 20: Ping preko celotnega omrežja (drugič)

Vidimo, da se že na razmeroma enostavnem primeru pokaže eden od problemov omrežij SDN – velika zakasnitev prvega paketa v podatkovnem toku. Problem, ki ga s pomočjo rešitve, predstavljene v tem delu, želimo občutno zmanjšati.

5.2.3 Krmilnik POX

POX [41] je odprtokodni krmilnik, napisan v programskem jeziku *Python*. Nastal je iz prav tako odprtokodnega krmilnika NOX [12, 37], ki je napisan v programskem jeziku C++. NOX je eden prvih krmilnikov SDN, izdelan za protokol *OpenFlow*. Ker je POX novejši, ima podprtih več omrežnih aplikacij, kot je npr. aplikacija za poizvedbo po topologiji omrežja. Oba podpirata samo protokol OF verzije 1.0.

Krmilnika POX in NOX sta namenjena spoznavanju in hitremu učenju tehnologije SDN. Oba se uporabljata za eksperimentiranje in izdelavo prototipnih rešitev na področju SDN. POX je tudi privzeto vključen v orodje *Mininet*.

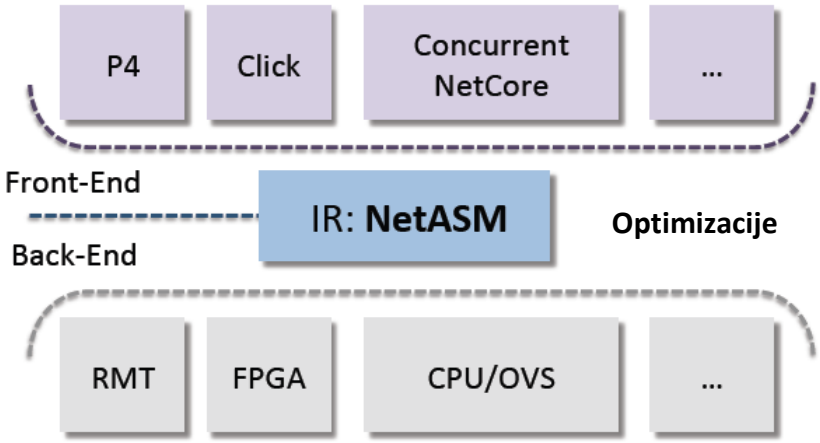
Za potrebe praktične implementacije tega dela je bil izbran krmilnik POX. V poglavju 5.3.2 je predstavljen program krmilnika POX z vključenimi potrebnimi prilagoditvami za izvajanje krmiljenja po vseh treh metodah.

5.2.4 Aplikacija NetASM

NetASM [23] je raziskovalni projekt univerze v Princtonu. Ponuja zanimivo alternativo trenutnemu stanju tehnologije SDN, ko se mora krmilnik točno zavedati, kakšen protokol podpirajo stikala podatkovne ravnine. NetASM predlaga nalaganje kompletnega programa za delovanje stikala v času povezovanja stikala s krmilnikom. Krmiljenje omrežja tako ni več odvisno od omejenega nabora akcij, ki jih ponuja npr. stikalo, zgrajeno po določeni verziji OF. S to rešitvijo lahko krmilnik stikalu vsakič spremeni nabor akcij, uporabo polj za primerjavo, uporabo tabel ipd. Skratka, programsko določi celotno arhitekturo stikala.

Ideja je torej v še večji programabilnosti omrežja, kot jo npr. ponuja že OF. Tudi drugi t. i. omrežni programski jeziki, npr.: *Click* [56], *Concurrent NetCore* [22], P4 [3] idr., udejanjajo to idejo. Vendar NetASM ponuja še nekaj več kot vmesni sloj med krmilnikom in stikali ponuja neke vrste prevajalno okolje za omrežja.

NetASM je, kot nakazuje že ime, strojni jezik za omrežja (ang. *network assembler*). Ponuja logično infrastrukturo, preko katere je mogoče vsak omrežni programski jezik prevesti in optimizirati za katerokoli strojno izvedbo stikala. Zgleduje se direktno po klasičnih prevajalnikih, pri katerih lahko program, napisan v kateremkoli programskem jeziku, prevedemo za izvajanje na katerikoli strojni arhitekturi oz. procesorju. Pri tem morata biti izpolnjena dva pogoja. Prvi je, da je za izbran programski jezik v prevajalniku na voljo prednji vmesnik (ang. *Front-End*). Drugi pa, da je za izbrano strojno arhitekturo v prevajalniku na voljo zadnji vmesnik (ang. *Back-End*). Slika 21 prikazuje vmesno vlogo aplikacije NetASM.



Slika 21: Vmesna vloga NetASM

NetASM poskrbi za abstraktno sliko podatkovne ravnine, ki je lahko implementirana na različne načine, npr.: z FPGA (ang. *Field-programmable Gate Array*), z RMT (ang. *Reconfigurable Match Tables*) [4] ali programsko – OVS (ang. *Open vSwitch*) [19, 43] idr.

Tako kot klasični prevajalniki tudi NetASM poleg prevajanja poskrbi tudi za optimizacijo kode. Definira lastni »strojni jezik«, ki vsebuje 23 primitivnih inštrukcij. Koda višje nivojskih programskih jezikov se najprej prevede v ta jezik, nato naprej v jezik za izvajanje na posameznem stikalu. Seveda je možno program napisati tudi direktno v strojnem jeziku NetASM. Ta možnost je bila izkoriščena tudi v našem primeru.

5.2.5 Slabosti aplikacije NetASM

Ideja aplikacije nima nikakršnih slabosti, edina slaba stran je, da koncept omrežnega prevajalnika (še) ni širše uporabljen in standardiziran. Spodnje slabosti tako izhajajo iz trenutne prototipne implementacije NetASM:

- Implementiran je v programskem jeziku *Python*[18], ki velja za enostaven in močan programski jezik, vendar hkrati tudi ne omogoča velikih zmogljivosti in ni optimalen glede porabe virov (npr. pomnilnika RAM).
- NetASM vsebuje prototipno podatkovno ravnino *NetASM Switch*[19], ki je tudi implementirana v jeziku *Python*. Delovanje te podatkovne ravnine oz. stikala je

[18] Obstaja tudi starejša implementacija v programskem jeziku *Haskell*.
[19] Trenutno v resnici edina podatkovna ravnina, ki jo prototipna izvedba NetASM podpira na svoji spodnji strani.

bistveno počasnejše kot v poglavju 5.2.2 predstavljen primer z uporabo privzete podatkovne ravnine v orodju *Mininet*. Hkrati vsako takšno stikalo porabi zelo veliko pomnilnika RAM, zato je izgradnja velikega omrežja, sestavljenega iz stikal NetASM, praktično nemogoča.

- Postopek programiranja je dokaj slabo (nepopolno) dokumentiran, slabo podprto pa je tudi razhroščevanje NetASM programske kode.

Kljub zapisanim slabostim prototipne implementacije je bil zaradi privlačnosti ideje, čiste radovednosti in želje po preizkušanju novosti NetASM izbran za izvedbo prototipa krmiljenja z razpršenim oddajanjem krmilnih sporočil. Celoten programski paket NetASM je v obliki izvorne kode na voljo na [57], v našem primeru je izbrana slika navideznega računalnika (poglavje 5.2.1) to kodo že vsebovala.

5.3 Implementacija razpršenega reakcijskega krmiljenja

Implementacijo v poglavju 4 predstavljene metode krmiljenja sestavljajo trije deli. Prvi je program *Mininet*, ki poskrbi za postavitev testnega omrežja. Drugi je program krmilnika POX, ki poskrbi za ustrezno krmiljenje, definirano v poglavju 4.5.1. In tretji, program stikala NetASM, ki poskrbi, da je stikalo zgrajeno ter da obdeluje krmilna sporočila in podatkovne pakete v skladu z definicijo v poglavju 4.5.2. Celotna implementacija v bistvu simulira minimalni potrebni del protokola *OpenFlow 1.0*[20] z zahtevanimi dodatki (poglavje 4.5.3).

Datoteke, ki sestavljajo implementacijo, so dodane v novo projektno mapo *eSDN* znotraj orodja NetASM. Celotna koda je na voljo tudi v dodatku (poglavje 8), naslednji razdelki pa opisujejo glavne funkcije in njihovo vlogo znotraj omenjenih treh delov.

5.3.1 Program *Mininet*

Program za postavitev testnega omrežja se nahaja v datoteki *eSDN_mininet.py*. Obsega eno glavno funkcijo *test_net()*, ki na začetku prebere vhodne parametre, podane v ukazni vrstici. Parametre nato uporabi za klic vgrajene funkcije *LinearTopo(),* ki zgradi linearno topologijo v velikosti, ki jo določajo parametri. Funkcija nato nastavi parametre stikal NetASM in zgradi testno omrežje iz pripravljene topologije, stikal in zunanjega, v tem trenutku še neznanega krmilnika. Nazadnje se startajo stikala, celotno omrežje ter *Mininet* CLI. Po izhodu iz vmesnika CLI se omrežje in stikala kontrolirano ustavijo.

5.3.2 Program krmilnika POX

Program krmilnika se nahaja v datoteki *eSDN_POX.py*. Krmilnik POX se vedno starta v funkciji *lunch()*. Funkcija najprej shrani vhodne parametre, podane v ukazni vrstici. Prvi parameter (*method*) pove, katero metodo krmiljenja bo krmilnik izvajal (Metodo I, II ali III). Ostala dva parametra definirata topologijo omrežja enako kot pri programu *Mininet*. Sledi priprava prostora za shranjevanje vpisanih tokovnih pravil na stikala. Nazadnje se krmilnik prijavi na sporočila, poslana s strani stikal. V našem primeru zadostuje prijava na sporočilo *ConnectionUp* (stikalo vzpostavi povezavo s krmilnikom) in sporočilo *VendorIn* – ekvivalent

[20] To je krmiljenje in posredovanje le na osnovi ciljnega naslova MAC. Primerjava ostalih polj in izvajanje ostalih akcij, predvidenih po standardu, za preizkus predloga ni potrebno.

sporočila *PacketIn*, le da stikalo NetASM uporablja specifičen format in zato uporablja sporočilo *VENDOR* (poglavje 2.6.2).

Vsako stikalo tako ob vzpostavitvi pošlje sporočilo *ConnectionUp*, kar proži izvajanje funkcije *_handle_ConnectionUp()* na krmilniku. Ta funkcija stikalu najprej pošlje celoten program, ki ga mora ta izvajati (poglavje 5.3.3). Šele, ko se program na stikalo naloži, ta lahko začne funkcionirati – sprejemati vpise v tabele in posredovati podatkovni promet na podlagi teh vpisov. Krmilnik v tokovno tabelo na novo povezanega stikala kot prvi vpis pošlje tokovno pravilo za posredovanje vsesmernih (ang. *broadcast*) paketov. Funkcija *_handle_ConnectionUp()* se s tem zaključi.

Ko krmilnik od stikala dobi sporočilo *VendorIn*, se proži funkcija *_handle_VendorIn()*. Ta funkcija na začetku preveri razlog, naveden v sporočilu, ki je praktično vedno *MATCH_TABLE_MISS*. Iz sporočila se nato izluščijo vsi potrebni podatki, s pomočjo katerih se preveri, ali je tip sporočila primeren za vpisovanje tokovnega pravila na stikalo.

Če smo z vhodnimi parametri izbrali krmiljenje po Metodi III, je postavljena zastavica *mrules*. V tem primeru se pokliče tudi funkcija za iskanje nadomestne sledi *_getAlternativeTrail()*. V kolikor se izračuna uporabna nadomestna sled *AT_dst*, funkcija vrne tudi identifikacijsko številko cepitvenega vozlišča *AT_sp_dpid*. Na to vozlišče (stikalo) se nato pošlje sporočilo za vpis pravila v tokovno tabelo (ekvivalent sporočila *FLOW_MOD*), na začetno stikalo[21] pa sporočilo *MCAST_RULE*, ki se pred tem sestavi preko klica *_mcastRuleMsgPack()*.

V kolikor smo z vhodnimi parametri izbrali krmiljenje po Metodi II, je postavljena zastavica *rules2all*. V tem primeru se preko funkcije *_getNodesOnPath* pridobi spisek vseh stikal na poti podatkovnega toka (razen začetnega stikala). V nadaljevanju se na vsa stikala s spiska pošlje sporočilo za vpis pravila v tokovno tabelo.

Ne glede na izbiro metode se v nadaljevanju na originalno stikalo pošlje prejeti paket ter sporočilo za vpis običajnega tokovnega pravila v tokovno tabelo. Obdelava sporočila *VendorIn* je s tem končana.

5.3.3 Program stikala NetASM

Program stikala se nahaja v datoteki *eSDN_netasm.py*. Predstavljen program je tisti del stikala NetASM, ki je predmet prenosa med krmilnikom in stikalom. Obstaja seveda tudi ogrodje stikala, ki poskrbi za:

- povezavo stikala s krmilnikom[22] in pošiljanje sporočila *ConnectionUp*,
- nalaganje programa, ki ga pošlje krmilnik,
- ustrezno obravnavo ostalih sporočil krmilnika,
- klic naloženega programa za vsak vhodni paket,
- posredovanje izhodnih paketov na vrata, določena znotraj programa in

[21] To je stikalo, od katerega je krmilnik prejel sporočilo *VendorIn*.
[22] Nastavitve za povezavo stikalo pridobi že ob startu, ki se proži v programu *Mininet* (poglavje 5.3.1).

- posredovanje sporočil, ustvarjenih znotraj programa, na krmilnik.

Tudi znotraj programa se kličejo samo funkcije, ki jih ponuja ogrodje. Te funkcije implementirajo primitivne strojne inštrukcije NetASM. Pripravljene funkcije in stil programiranja programa je prilagojen tako, da je napisana koda videti kot običajna strojna koda. Kot rečeno (poglavje 5.2.4), je prototipna izvedba NetASM, implementirana v programskem jeziku *Python*, zato je tudi koda našega programa v resnici običajna *Python* koda.

Program vsebuje eno samo funkcijo, ki mora biti poimenovana *main()*. Opozoriti velja, da v tem primeru ne gre za strogo proceduralno izvajanje kode, saj je namen, da se koda preslika v strojno opremo, kjer so možne tudi različne stopnje paralelizma. Tako se začetni del funkcije *main()*, kjer se preko deklaracij kreirajo tabele, izvede samo enkrat – ob nalaganju programa.

Glavna koda programa, ki se procesira za vsak podatkovni paket, se začne ob oznaki *### Code ###*. Tu se najprej definirajo in preberejo iz paketa vsa primerjalna polja (V primeru, da gre za paket *MCAST_RULE*, se prebere še nekaj dodatnih polj). Nadaljevanje izvajanja je ločeno na procesiranje paketov *MCAST_RULE* in procesiranje podatkovnih paketov.

Procesiranje paketa *MCAST_RULE* je v skladu z diagramom poteka na Sliki 12 (poglavje 4.3.2) ter dodatki v poglavju 4.5.2. Najprej se preveri, če vpis ne obstaja že v osnovni tokovni tabeli *match-table*. V primeru obstoja se izvajanje takoj zaključi (gre za cepitveno vozlišče). V nasprotnem primeru se vsebina vpiše v tabelo *MR_match_table*, paket se posreduje v smeri nadomestne sledi.

Procesiranje podatkovnih paketov je v skladu z diagramom poteka na Sliki 14 (poglavje 4.3.2) ter dodatki v poglavju 4.5.2. Koda tu na začetku deklarira nekaj polj za začasne podatke. Nato se prisotnost ustreznega tokovnega pravila preveri najprej v tabeli *match-table*, v primeru odsotnosti pa še v tabeli *MR_match_table*. Če ustreznega pravila tudi v drugi tabeli ni, se paket posreduje na krmilnik, izvajanje pa se konča. V kolikor je pravilo v eni od tabel prisotno, se iz te tabele prebere indeks akcije, ki se uporabi za branje akcije iz tabele akcij. Nazadnje se paket posreduje v skladu s prebrano akcijo.

5.4 Zagon okolja

Vsakega od programov, opisanih v prejšnjem poglavju je potrebno zagnati v svojem terminalnem oknu SSH. Najprej poženemo program *Mininet* v prvem oknu. Program po postavitvi omrežne topologije izpiše ukazno vrstico za zagon aplikacije NetASM, ki jo prenesemo v drugo terminalno okno in izvedemo. V tretjem oknu nato izvedemo še vrstico za zagon krmilnika POX. NetASM in POX se nato povežeta in krmilnik na vsa stikala v omrežju naloži program NetASM – postopek traja nekaj časa (odvisno od števila stikal), preko izpisov v drugem oknu pa lahko spremljamo dogajanje. Ko so vsa stikala naložena, lahko v prvem oknu začnemo z izvajanjem konzolnih ukazov *Mininet*.

Za zagon našega okolja s topologijo s Slike 18, je tako potrebno izvesti naslednje ukazne vrstice:

- Prvo terminalno okno – *Mininet*:

```
sudopy ~/netasm/netasm/examples/eSDN/eSDN_mininet.py --cli --switches=7
--hosts_per_switch=3
```

- Drugo terminalno okno – NetASM (prenos ukaza iz izpisa v prvem oknu):

```
sudopy python /home/vagrant/pox/pox.py --no-openflow
netasm.back_ends.soft_switch.datapath --address=127.0.0.1 --port=6633
--dpid=0000000000000001 --policy= --ports=s1-eth1,s1-eth2,s1-eth3,s1-
eth4 netasm.back_ends.soft_switch.datapath --address=127.0.0.1
--port=6633 --dpid=0000000000000002 --policy= --ports=s2-eth1,s2-
eth2,s2-eth3,s2-eth4,s2-eth5 netasm.back_ends.soft_switch.datapath
--address=127.0.0.1 --port=6633 --dpid=0000000000000003 --policy=
--ports=s3-eth1,s3-eth2,s3-eth3,s3-eth4,s3-eth5
netasm.back_ends.soft_switch.datapath --address=127.0.0.1 --port=6633
--dpid=0000000000000004 --policy= --ports=s4-eth1,s4-eth2,s4-eth3,s4-
eth4,s4-eth5 netasm.back_ends.soft_switch.datapath --address=127.0.0.1
--port=6633 --dpid=0000000000000005 --policy= --ports=s5-eth1,s5-
eth2,s5-eth3,s5-eth4,s5-eth5 netasm.back_ends.soft_switch.datapath
--address=127.0.0.1 --port=6633 --dpid=0000000000000006 --policy=
--ports=s6-eth1,s6-eth2,s6-eth3,s6-eth4,s6-eth5
netasm.back_ends.soft_switch.datapath --address=127.0.0.1 --port=6633
--dpid=0000000000000007 --policy= --ports=s7-eth1,s7-eth2,s7-eth3,s7-
eth4 --ctl_port=7791
```

- Tretje terminalno okno (POX) – tri možnosti:

1. Simulacija omrežja s krmiljenjem po Metodi I:

```
sudopy ./pox/pox.py netasm.examples.eSDN.eSDN_pox --method=1
--switches=7 --hosts_per_switch=3
```

2. Simulacija omrežja s krmiljenjem po Metodi II:

```
sudopy ./pox/pox.py netasm.examples.eSDN.eSDN_pox --method=2
--switches=7 --hosts_per_switch=3
```

3. Simulacija omrežja s krmiljenjem po Metodi III:

```
sudopy ./pox/pox.py netasm.examples.eSDN.eSDN_pox --method=3
--switches=7 --hosts_per_switch=3
```

6 REZULTATI SIMULACIJ

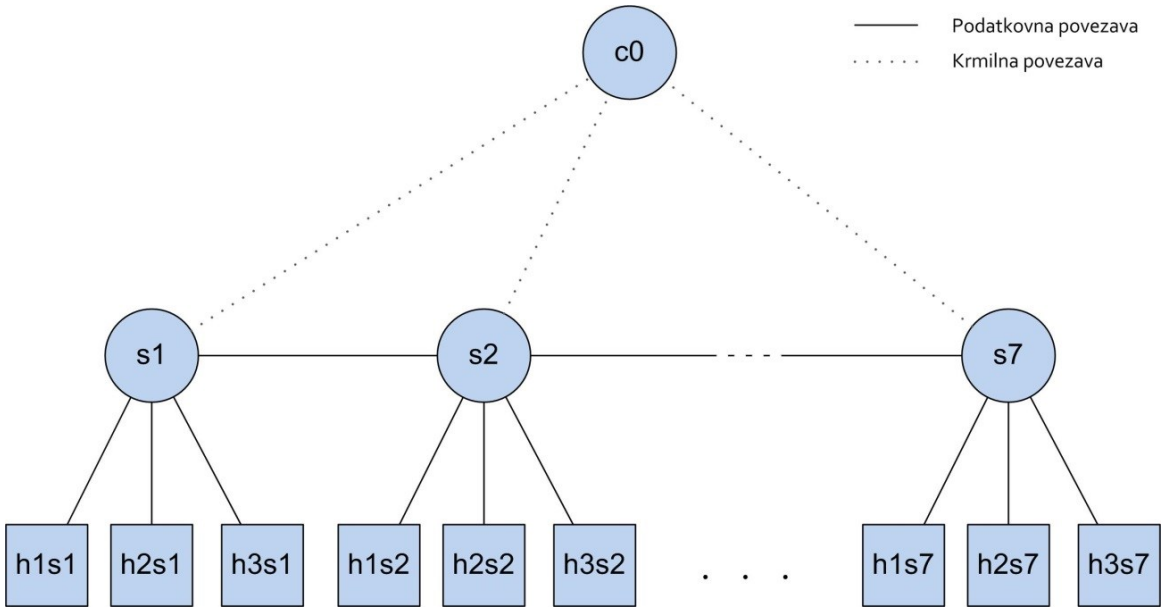
V tem poglavju so predstavljeni rezultati testnih meritev, opravljenih na simulacijskem okolju, opisanem v poglavju 5.2. Meritve sledijo ciljem, postavljenim v poglavju 5.1. Njihova izvedba temelji na programih, predstavljenih v poglavju 5.3. Simulacijsko okolje se zažene s pomočjo ukaznih vrstic, zapisanih v poglavju 5.4.

V nadaljevanju so podrobneje predstavljene vse karakteristike in privzete predpostavke simulacijskega okolja. Sledijo podrobni opisi vseh izvedenih testov skupaj s predstavitvijo rezultatov. Nazadnje je podana analiza rezultatov in končno ovrednotenje predlagane metode z razpršenim reakcijskim krmiljenjem (Metoda III).

6.1 Predpostavke simulacijskega okolja

6.1.1 Topologija omrežja

Zaradi velike porabe virov s strani aplikacije NetASM (poglavje 5.2.5) je topologija testnega omrežja omejena na 7 stikal. Metoda III (poglavje 4.3) je najbolj učinkovita na čim daljših povezavah, na katerih je po možnosti vedno na voljo že obstoječa povezava (preko celotne dolžine), ki se uporabi kot nadomestna sled[23]. Posledično smo vseh 7 razpoložljivih stikal postavili v vrsto in zgradili enako linearno topologijo, kot smo jo spoznali že v poglavju 5.2.2 (Slika 21).



Slika 21: Topologija simuliranega testnega omrežja

[23] Kot že zapisano, se to lahko zagotoviti z ustrezno postavitvijo začetnih, proaktivno vzpostavljenih in tokovnih povezav.

Na Sliki 21 vozlišča z oznako s_i predstavljajo stikala, c_0 označuje krmilnik, oznake $h_i s_j$ predstavljajo gostitelje. Polne črte ponazarjajo podatkovne, točkaste pa krmilne povezave.

Učinkovitost krmiljenja po Metodi III ni v ničemer odvisna od topologije omrežja razen od dolžine posameznih povezav. S topologijo s Slike 21 lahko preverimo učinkovitost na vseh povezavah, dolgih do sedem stikal. Dobljene rezultate lahko aproksimiramo na omrežje poljubne topologije, potrebujemo le podatek o povprečni dolžini povezav v izbranem omrežju.

6.1.2 Ostale predpostavke

Poleg zgoraj definirane topologije, izvajanje simulacij sloni še na naslednjih karakteristikah in predpostavkah:

1. Simulacijsko okolje je postavljeno na osebnem računalniku z dvojedrnim procesorjem s taktom 3 GHz in 8 GB RAM pomnilnika.

2. Da se za potrebe simulacije zagotovi čim več virov, poskrbimo, da med simulacijo na računalniku niso aktivni nepotrebni programi.

3. Simulacije izvajamo na popolnoma izoliranem okolju. V postavljenem omrežju ni drugega podatkovnega prometa kot tistega, ki ga ustvarijo naši testi. Edina izjema so periodična sporočila TCP na povezavi med krmilnikom in stikali, namenjeni preverjanju povezljivosti in stanja teh naprav.

4. V simulacijskem okolju so povezave med krmilnikom in stikali popolnoma ločene od povezav, namenjenih prenosu podatkovnih paketov. V realnem okolju takšne dodatne povezave praktično nikoli niso na voljo. Krmilna sporočila si v realnih omrežjih tako delijo razpoložljivo pasovno širino s podatkovnim prometom. V primeru Metode III za sporočilo *MCAST_RULE* velja enako tudi v simulacijskem okolju. Morebitne razlike zaradi obstoja ločene povezave za vsa ostala sporočila pri analizi rezultatov zanemarimo.

5. Metodo III želimo ovrednotiti na podlagi primerjav z Metodo I in Metodo II. Vse meritve razen dokaza koncepta so tako opravljene s simuliranjem vseh treh metod krmiljenja.

6. Zaradi že omenjene neoptimalne izvedbe aplikacije NetASM so časi *ping* neobičajno dolgi. Kot rečeno, s simulacijskimi meritvami izvajamo primerjavo različnih metod krmiljenja, zato absolutna vrednost izmerjenih časov nima posebnega pomena.

7. Eden od razlogov dolgih časov *ping* je tudi ta, da se aplikacija NetASM nahaja v uporabniškem prostoru (ang. *user space*) operacijskega sistema. Zaradi kopiranj paketov pri prehodu med uporabniškim in jedrnim prostorom (ang. *kernel space*) se precej podaljša čas obdelave paketov. Primer v poglavju 5.2.2 npr. uporablja v *Mininetu* privzeti *OpenFlow kernel switch*, ki se nahaja v jedru in tako lahko hitreje obdeluje pakete.

8. Pri izvajanju meritev z ukazom *ping* vedno dodamo opcijski parameter *-Ai3*, ki pomeni, da je interval pošiljanja paketov *ping* 3 sekunde ter da se v primeru hitrejšega

odziva interval lahko avtomatsko skrajša. S tem preprečimo morebitno neuspešnost meritve zaradi (pre)dolgih časov *ping*.

9. Posledica velikih zakasnitev na stikalih je slaba prepustnost. Izmerjena prepustnost s pomočjo ukaza *iprf* znotraj *Minineta* je tako preko enega stikala (npr. med *h1s1* in *h2s1*) približno 100 Kb/s. Meritev preko vseh sedmih stikal (npr. med *h1s1* in *h1s7*) pokaže le še dobrih 10 Kb/s. Oba podatka veljata za vse tri metode krmiljenja. Zaradi slabe prepustnosti simulacijskega omrežja meritve izvajamo tako, da v omrežju ni hkrati prisotnih veliko število paketov. Eden od ukrepov, ki to zagotavlja, je v prejšnji točki predstavljen parameter *-Ai3*.

10. *Mininet* omogoča tudi omejevanje pasovne širine in nastavitev zakasnitev na povezavah. Zaradi že tako slabih zmogljivosti stikal teh možnosti pri simulaciji ne izkoriščamo. Rezultat meritve prepustnosti povezave med dvema stikaloma (npr. *s1* in *s2*), je tako slabih 10 Gb/s. Enako hitra je tudi povezava med stikali in krmilnikom.

11. Začetno izvajanje protokola ARP je enako pri vseh treh metodah in nima vpliva na primerjalne rezultate. Da ta proces ne bi vplival na meritev pri gradnji topologije v programu *Mininet*, s pomočjo parametra *autoStaticArp=True* vključimo statični vpis vseh vnosov v tabelo ARP.

12. Pri izvajanju meritev je bilo zaznano, da je dobljen začetni čas prvega ukaza *ping* po zagonu stikal vedno precej slabši od enakovrednih ponovitev v nadaljevanju. Na primer, če je prvi *ping* v seriji meritev med *h1s1* in *h1s7*, zatem pa sledi *ping* med *h2s1* in *h2s7*, je drugi začetni čas vedno za okoli 30 odstotkov boljši od prvega, čeprav bi za Metodo I in Metodo II pričakovali zelo podobna časa. Predvidevamo, da je to posledica izvajanja simulacij na virtualnem okolju, v katerem se na začetku programska koda naloži v medpomnilnik, za to pa je potreben dodaten čas. Časov prvega *ping-a* v seriji meritev zato ne upoštevamo v statistiki rezultatov.

13. V primeru Metode III, v prejšnji točki obravnavan prvi *ping*, izkoristimo za vzpostavitev prve tokovne povezave preko vseh stikal v obe smeri. Ti dve povezavi se v nadaljevanju uporablja kot nadomestno sled pri ostalih povezavah. Prvi *ping* se tako, v primeru Metode III, uporablja kot nadomestek v teoriji predvidenih proaktivno vzpostavljenih začetnih tokovnih povezav.

14. V teoriji krmilnik pri Metodi III prvemu stikalu na poti odgovori z dvema sporočiloma, *MCAST_RULE* in *PACKET_OUT* (Slika 11 v poglavju 4.3.1). Zaradi enostavnejše implementacije se v našem primeru poleg teh dveh sporočil na prvo stikalo pošlje dodatno sporočilo *FLOW_MOD*. Posledično pričakujemo, da bo to vplivalo na za odtenek slabše izmerjene rezultate predlagane metode.

6.2 Dokaz koncepta

Da se prepričamo, ali metoda z razpršenim reakcijskim krmiljenjem deluje, na postavljenem simulacijskem okolju najprej preverimo, če je možno vzpostaviti povezljivost med vsemi gostitelji v omrežju. V ta namen v ukazni vrstici *Mininet* izvedemo ukaz *pingall*. Še pred izvedbo tega ukaza izvedemo običajen (prvi) *ping* preko celotnega omrežja. S tem se preko omrežja vzpostavita dve osnovni tokovni povezavi (v vsako stran po ena), ki bosta služili kot

nadomestna sled vsem ostalim. Slika 22 prikazuje izvedbo obeh omenjenih ukazov na našem testnem omrežju (Slika 21), krmiljenem po Metodi III.

```
mininet> h1s1 ping -c1 -W3 h1s7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=1 ttl=64 time=2899 ms

--- 10.0.0.7 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2899.845/2899.845/2899.845/0.000 ms
mininet>
mininet> pingall
*** Ping: testing ping reachability
h1s1 -> h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h1s2 -> h1s1 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h1s3 -> h1s1 h1s2 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h1s4 -> h1s1 h1s2 h1s3 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h1s5 -> h1s1 h1s2 h1s3 h1s4 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h1s6 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h1s7 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h2s1 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h2s2 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h2s3 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h2s4 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h2s5 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h2s6 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h2s7 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h3s1 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s2 h3s3 h3s4 h3s5 h3s6 h3s7
h3s2 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s3 h3s4 h3s5 h3s6 h3s7
h3s3 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s4 h3s5 h3s6 h3s7
h3s4 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s5 h3s6 h3s7
h3s5 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s6 h3s7
h3s6 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s7
h3s7 -> h1s1 h1s2 h1s3 h1s4 h1s5 h1s6 h1s7 h2s1 h2s2 h2s3 h2s4 h2s5 h2s6 h2s7 h3s1 h3s2 h3s3 h3s4 h3s5 h3s6
*** Results: 0% dropped (420/420 received)
mininet>
```

Slika 22: Dokaz koncepta z ukazom pingall

Vidimo, da je s krmiljenjem po predlagani metodi možno vzpostaviti povezljivost med vsemi gostitelji v omrežju. To pomeni, da je bil preizkus koncepta za Metodo III uspešen.

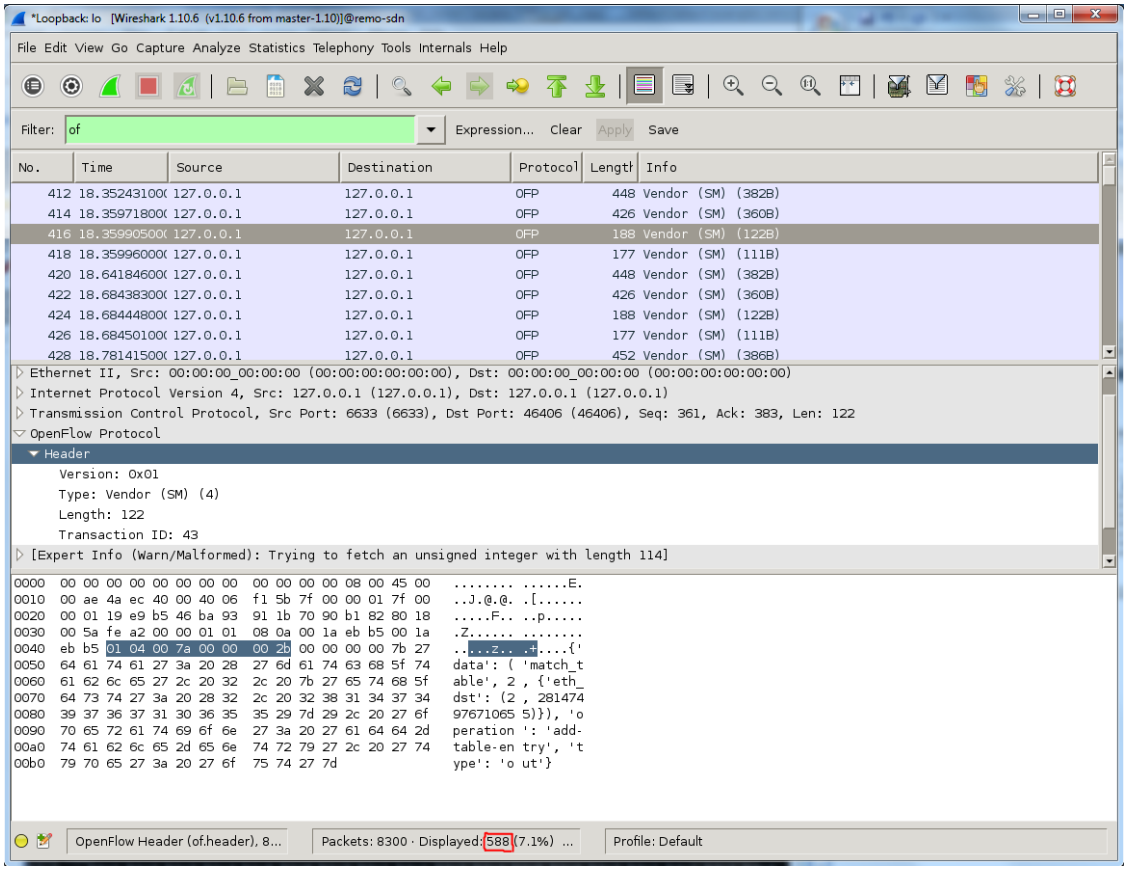
6.3 Primerjava števila krmilnih sporočil

Prednosti metode z razpršenim reakcijskim krmiljenjem (poglavje 4.6.1) posredno in neposredno izhajajo iz trditve, da je povezljivost možno vzpostaviti z izmenjavo manjšega števila sporočil med krmilnikom in stikali. Preverjanje potrebnega števila krmilnih sporočil za vzpostavitev povezav smo za vse tri metode krmiljenja izvedli preko ukaza *pingall*. Pri tem smo zajeli vsa krmilna sporočila, ki so bila sprejeta ali poslana s strani krmilnika, za vzpostavljanje vseh možnih povezav v testni topologiji s Slike 21.

Med izvajanjem testov je bilo vključeno zajemanje prometa s programom *Wireshark*. Zajemanje je bilo vključeno na vmesniku *lopback*, preko katerega v simulacijskem okolju poteka komunikacija med krmilnikom in stikali. Vključeno je bilo tudi filtriranje paketov po protokolu OF, saj kljub nekoliko drugačnemu formatu, vsa sporočila v našem okolju vsebujejo glavo OF. Ker ostala vsebina ni standardna, *Wireshark* vsebine, ki sledi glavi OF, ne more interpretirati. Filtriranje je vključeno, ker se pri zajemanju na omenjenem vmesniku zajamejo tudi že omenjeni paketi za preverjanje povezljivosti med krmilnikom in stikali.

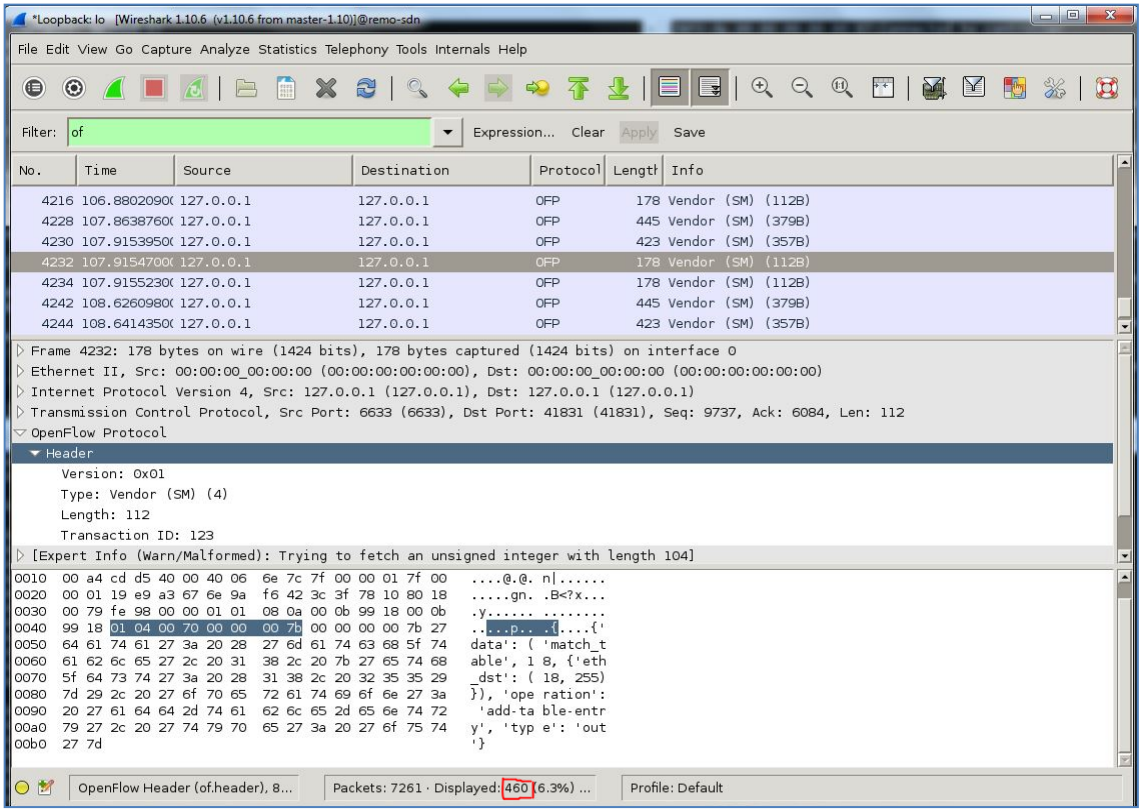
S pomočjo programa *Wireshark* smo tako zajeli in s filtrom izločili vsa krmilna sporočila, izmenjana v času testa *pingall*. Po končani meritvi vseh treh metod krmiljenja naredimo

primerjavo števila preko filtra zajetih paketov. Dobljena razlika direktno predstavlja izboljšavo posameznega krmiljenja.

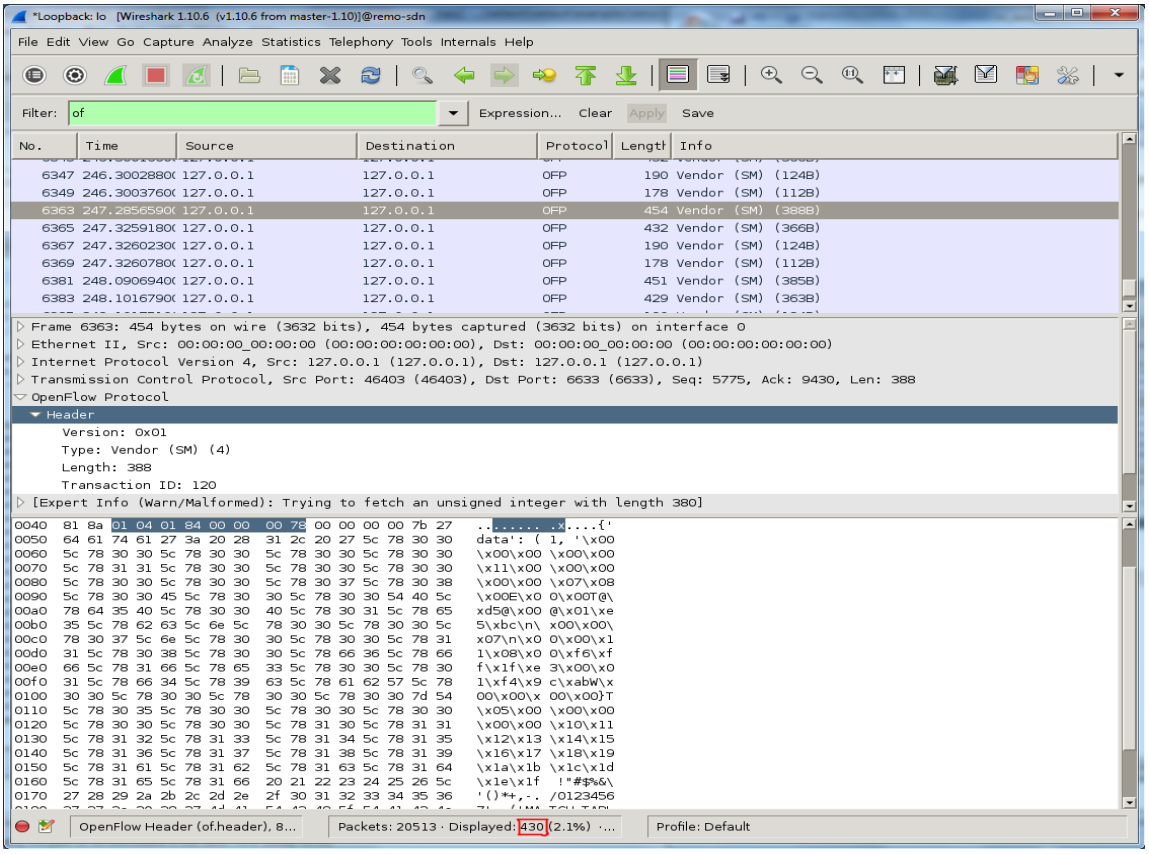


Slika 23: Število zajetih sporočil med testom pingall (Metoda I)

S Slike 23 je razvidno stanje po izvedbi testa *pingall* za osnovno metodo krmiljenja – število vseh zajetih sporočil je 588. Slika 24 prikazuje stanje po končanem enakem testu za Metodo II, Slika 25 pa za Metodo III.



Slika 24: Število zajetih sporočil med testom pingall (Metoda II)



Slika 25: Število zajetih sporočil med testom pingall (Metoda III)

Meritve pokažejo, da je potrebno število krmilnih sporočil za vzpostavitev vseh povezav našega omrežja v primeru Metode I 588, v primeru Metode II 460 in v primeru Metode III 430. Tabela 1 povzema dobljene rezultate.

Metoda krmiljenja	Št. krmilnih sporočil med izvedbo ukaza *pingall*
Metoda I	588
Metoda II	460
Metoda III	430

Tabela 1: Primerjava števila krmilnih sporočil

Vidimo, da je pri predlagani metodi z razpršenim reakcijskim krmiljenjem v našem primeru potrebnih skoraj 30 odstotkov manj krmilnih sporočil kot pri krmiljenju po Metodi I ter približno 7 odstotkov manj sporočil kot pri krmiljenju po Metodi II. Razlika bi bila še nekoliko večja brez uporabe dodatnega sporočila, omenjenega v predpostavki 14 v poglavju 6.1.2. Še večja razlika bi bila na omrežju z daljšo povprečno dolžino poti. V našem omrežju s sedmimi stikali v linearni topologiji je povprečna dolžina poti 3,5 stikal, to je v vseh pogledih majhna dolžina, dobljena razlika (7 %) pa je v tem oziru relativno velika.

Opravljena simulacija je pokazala, da predlagana metoda krmiljenja izpolnjuje cilje glede zmanjšanja števila sporočil, potrebnih za vzpostavljanje tokovnih povezav. Posledično predvidevamo, da je tudi obremenitev krmilnika zmanjšana v približno enaki meri.

6.4 Primerjava časov

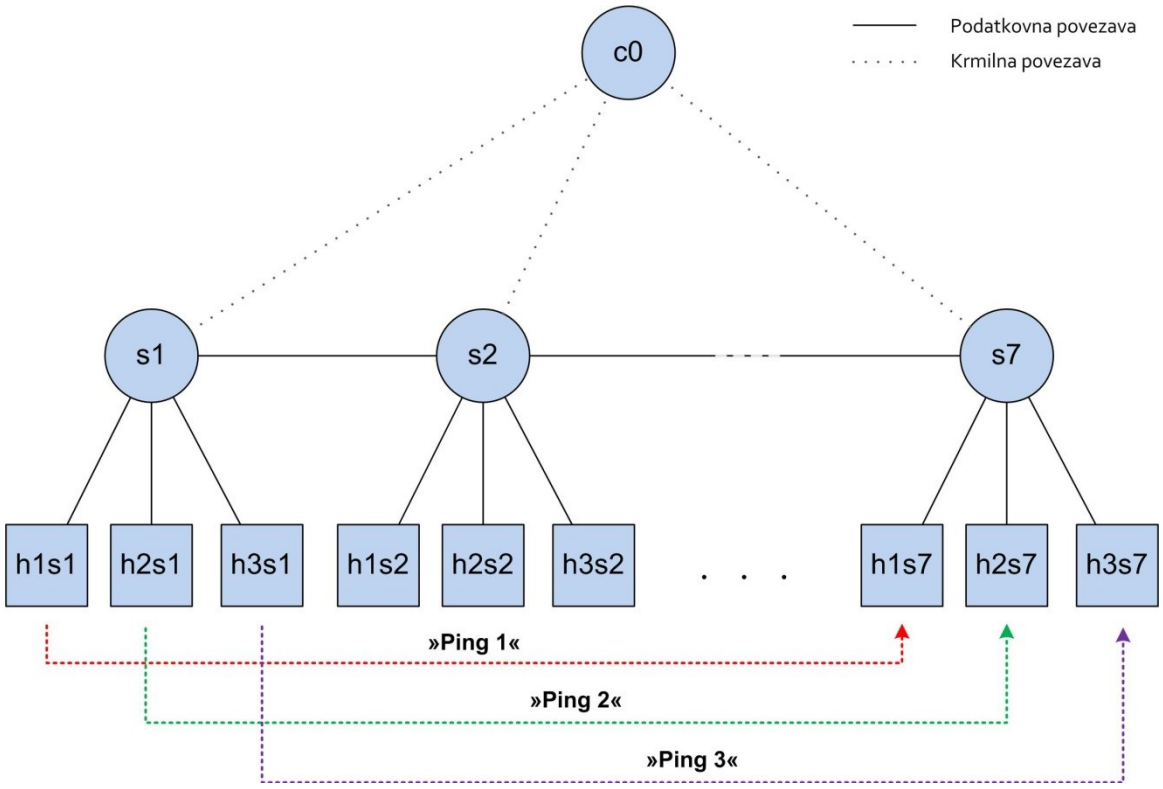
Z namenom vrednotenja Metode III nas poleg zmanjšanja števila sporočil, potrebnih za vzpostavitev tokovnih povezav, zanima tudi primerjava dveh časov:

- primerjava časa, potrebnega za vzpostavitev tokovne povezave in
- primerjava časa oz. zakasnitve pri posredovanju paketov na stikalih.

Tako kot pri primerjavi števila sporočil, opisani v prejšnjem poglavju, tudi primerjavo časov opravimo med vsemi tremi metodami krmiljenja. Kot posledico najmanjšega števila sporočil, potrebnih za vzpostavitev podatkovnih poti pri krmiljenju po Metodi III, pričakujemo pri isti metodi tudi najmanjše zakasnitve pri vzpostavljanju tokovnih povezav.

Test primerjave časov pri izvedbi uporablja tri ukaze *ping* med različnimi gostitelji na skrajnih točkah topologije, kot kaže Slika 26. Pri vseh treh metodah krmiljenja izvedemo serijo prikazanih ukazov *ping*, zajamemo rezultate ter pobrišemo vse vzpostavljene povezave – to pomeni brisanje vseh tabel na stikalih. Ko so vsa stikala postavljena na izhodiščno stanje, lahko serijo treh ukazov *ping* ponovimo. Vsako izhodiščno stanje predstavlja situacijo, opisano pod predpostavko 12 v poglavju 6.1.2, zato časa prvega ukaza *ping* nikoli ne shranimo med zajete rezultate.

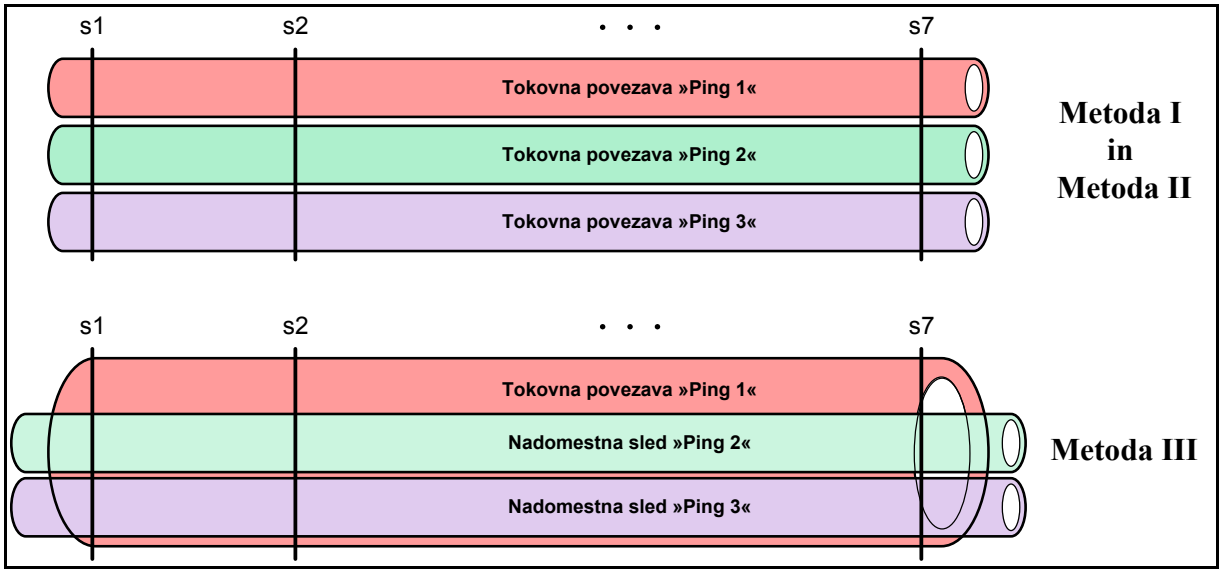
Meritve časov izvajamo samo na povezavah dolžine 7. S tem pričakujemo, da se bo v danem omrežju Metoda III izkazala najbolje možno (poglavje 6.1.1). Dobljeni rezultati bodo tako relevantni za omrežja poljubnih topologij, pri katerih je povprečna dolžina tokovnih povezav enaka 7.



Slika 26: Potek merjenja časov na testni topologiji

Kot predvideva predpostavka 13, izvedba ukaza »Ping 1« s Slike 26 v primeru Metode III poskrbi za vzpostavitev prve tokovne povezave preko vseh stikal v obe smeri. Ti povezavi ostala dva ukaza *ping* izkoristita kot nadomestno sled (Slika 27).

Pri drugih dveh metodah krmiljenja (Metoda I in Metoda II) vsak od ukazov *ping* vzpostavi svojo tokovno povezavo, saj v njunih primerih ne gre za pouporabo predhodno vzpostavljenih sledi. Slika 27 prikazuje primerjavo vzpostavljenih povezav pri različnih metodah krmiljenja.



Slika 27: Primerjava vzpostavljenih povezav po metodah krmiljenja

Vsak od ukazov *ping* proži pošiljanje več zaporednih paketov *ping* (Slika 28). Prvi od teh paketov vzpostavi tokovno povezavo na enega od načinov s Slike 27 v odvisnosti od uporabljene metode krmiljenja. Vsi naslednji paketi *ping* potujejo po že vzpostavljeni povezavi in zato nikoli ne pridejo v obdelavo na krmilnik. Obhodni časi vseh teh paketov so zato precej manjši kot čas obhoda prvega paketa *ping*.

Kot rezultat izvajanja ukaza *ping* dobimo čas, potreben za vzpostavitev tokovne povezave. Obhodni čas prvega paketa *ping* poleg zakasnitve pri posredovanju na stikalih vsebuje še čas, potreben za vzpostavitev obeh tokovnih povezav (po eno v vsaki smeri). Kot rečeno, vsi naslednji paketi *ping* potujejo po že vzpostavljenih tokovnih povezavah, zato njihov čas vsebuje samo zakasnitve pri posredovanju na stikalih. Če od časa prvega paketa odštejemo povprečni čas paketov, ki sledijo, dobimo čas, potreben za vzpostavitev obeh tokovnih povezav. Za izračun omenjenega povprečnega časa uporabimo obhodne čase petih paketov *ping* – torej čas od drugega do šestega paketa. Vsakega od ukazov *ping*, zato, poleg v predpostavki 9 omenjenega parametra *-Ai3*, zaženemo še s parametrom *-c6*, ki omeji izvajanje na 6 paketov *ping*.

Pri izvedbi opisanega postopka lahko odčitamo oba časa, omenjena na začetku poglavja. Slika 28 prikazuje vzorčno izvedbo testa z ukazom *ping* za primer krmiljenja po Metodi III.

```
mininet> h3s1 ping -Ai3 -c6 h3s7
PING 10.0.0.21 (10.0.0.21) 56(84) bytes of data.
64 bytes from 10.0.0.21: icmp_seq=1 ttl=64 time=1603 ms
64 bytes from 10.0.0.21: icmp_seq=2 ttl=64 time=1288 ms
64 bytes from 10.0.0.21: icmp_seq=3 ttl=64 time=1264 ms
64 bytes from 10.0.0.21: icmp_seq=4 ttl=64 time=1161 ms
64 bytes from 10.0.0.21: icmp_seq=5 ttl=64 time=1225 ms
64 bytes from 10.0.0.21: icmp_seq=6 ttl=64 time=1364 ms
```

Slika 28: Izvedba testa ping na simulacijskem omrežju, krmiljenem po Metodi III

Iz vzorčnega testa *ping* na Sliki 28 lahko razberemo naslednje podatke:

- celotni čas prvega paketa *ping* ($\Delta p1$) je 1603 ms,
- povprečni čas od drugega do šestega paketa *ping* oz. zakasnitev pri posredovanju paketov na stikalih ($\Delta p2..6$) je 1260 ms in
- čas, potreben za vzpostavitev tokovne povezave ($\Delta ts = \Delta p1 - \Delta p2..6$) je 343 ms.

Tabela 2 prikazuje rezultate opravljenih simulacij z izračunanimi zakasnitvami in povprečnimi vrednostmi časov za vse tri metode krmiljenja.

Krmiljenje po Metodi I				Krmiljenje po Metodi II				Krmiljenje po Metodi III			
No.	Δp1 [ms]	Δp2..6 [ms]	Δts [ms]	No.	Δp1 [ms]	Δp2..6 [ms]	Δts [ms]	No.	Δp1 [ms]	Δp2..6 [ms]	Δts [ms]
1	1934	1179	755	1	1545	1180	365	1	1506	1147	359
2	1880	1164	716	2	1414	1188	226	2	1603	1260	343
3	1676	1055	621	3	1411	1259	152	3	1786	1322	464
4	1565	1136	429	4	1286	1133	153	4	1656	1304	352
5	1747	1232	515	5	1324	1157	167	5	1659	1232	427
6	1838	1130	708	6	1381	1217	164	6	1602	1322	280
7	1943	1108	835	7	1527	1212	315	7	1535	1245	290
8	1603	1114	489	8	1418	1170	248	8	1633	1288	345
9	1994	1213	781	9	1325	1159	166	9	1694	1289	405
10	1557	1117	440	10	1319	1168	151	10	1698	1391	307
11	1831	1150	681	11	1342	1180	162	11	1692	1237	455
12	1725	1164	429	12	1293	1189	104	12	1620	1235	385
13	1686	1189	515	13	1436	1164	272	13	1651	1333	318
14	1923	1191	732	14	1399	1238	161	14	1839	1338	501
15	1765	1220	545	15	1311	1157	154	15	1749	1348	401
Povp.	**1777,80**	**1157,47**	**620,33**		**1382,07**	**1184,73**	**197,33**		**1661,53**	**1286,07**	**375,47**

Tabela 2: Tabela rezultatov simulacij vseh treh metod krmiljenja

Legenda stolpcev v Tabeli 2:

- No. – zaporedna številka.
- Δp1 – celotni čas prvega paketa *ping*.
- Δp2..6 – povprečni čas od drugega do šestega paketa *ping* (zakasnitev pri posredovanju paketov na stikalih).
- Δts – čas, potreben za vzpostavitev tokovne povezave, izračunan po enačbi: Δts = Δp1 – Δp2..6.
- Povp. – povprečni rezultat.

Iz povprečnih vrednosti v spodnji vrstici tabele na Sliki 29 lahko razberemo naslednja razmerja:

- Čas, potreben za vzpostavljanje tokovnih povezav, je v primeru krmiljenja po predlagani Metodi III sicer občutno krajši kot v primeru krmiljenja po Metodi I, vendar vseeno precej daljši kot v primeru krmiljenja po Metodi II.
- Zakasnitev pri posredovanju paketov na stikalih je v primeru krmiljenja po Metodi I in Metodi II primerljiva, v primeru Metode III pa nekoliko večja.
- Tudi celotni čas prvih paketov *ping* je najboljši v primeru krmiljenja po Metodi II.

Proti pričakovanjem se je pri merjenju časov v simulacijskem okolju kot najučinkovitejša izkazala Metoda II (poglavje 3.2). Razlog je v razliki med prepustnostjo krmilnih povezav in prepustnostjo stikal v simulacijskem omrežju, podrobnejša razlaga sledi v naslednjem poglavju.

6.5 Analiza rezultatov

Vse opravljene meritve so bile izvedene v simulacijskem okolju, ki vključuje omrežje enostavne topologije. Enostavnost topologije nima direktne povezave z delovanjem katerekoli od simuliranih metod krmiljenja, saj je prepoznavanje topologije in njena ustrezna obravnava naloga višje ležečih omrežnih storitev znotraj krmilnika (poglavje 2.1). Izbrano krmiljenje le uporablja izračune te storitve. Dobljene rezultate je tako z ustrezno preslikavo možno uporabiti v poljubnem omrežju.

Za predlagano metodo z razpršenim reakcijskim krmiljenjem je bil najprej izveden preizkus koncepta. Na postavljeni topologiji smo pokazali, da je metoda funkcionalno uporabna. Glede na zapis zgoraj trdimo, da to velja za omrežje poljubne topologije.

Druga meritev je zajemala primerjavo potrebnega števila krmilnih sporočil za vzpostavitev vseh možnih tokovnih povezav v testnem omrežju. To število smo izmerili za krmiljenje z vsemi tremi metodami in opravili primerjavo rezultatov. Po pričakovanjih smo pri predlagani metodi z razpršenim reakcijskim krmiljenjem dobili najmanjše število. Tudi v tem primeru trdimo, da je ugoden rezultat možno preslikati na omrežje poljubne topologije, ne nazadnje smo to tudi teoretično pokazali v poglavju 4.3.3.

Najmanjše število prejetih in oddanih krmilnih sporočil, pomeni tudi najmanjšo obremenitev krmilnika. Zato predlagana metoda zmanjšuje glavne slabosti omrežij SDN.

Glavno meritev na simuliranem omrežju predstavlja merjenje časov vzpostavljanja tokovnih povezav in zakasnitev pri posredovanju paketov na stikalih. Proti pričakovanju pri predlagani metodi nismo izmerili najboljšega časa vzpostavljanja povezave. Razlog je v slabi zmogljivosti stikal NetASM. Slabi prepustnosti in zakasnitvam omenjenim v predpostavkah 6., 7. in 9. (poglavje 6.1.2) je pri Metodi III poleg podatkovnih paketov podvrženo tudi sporočilo *MCAST_RULE*, ki do sosednjih stikal pride počasneje kot množica krmilnih sporočil *FLOW_MOD* pri Metodi II. Ta sporočila potujejo od krmilnika direktno proti stikalom po bistveno hitrejših krmilnih povezavah, zato se Metoda II po meritvah izkaže kot najučinkovitejša.

Kot je že zapisano v predpostavki 4, se sporočila v realnih omrežjih vedno pošiljajo po istih fizičnih povezavah. V teh primerih zato ne more priti do nikakršnih razlik v hitrosti različnih sporočil. V končni fazi to pomeni, da bi v realnem okolju ali pa v primernejšem simulacijskem okolju vseeno dobili krajši čas vzpostavljanja povezave s krmiljenjem po Metodi III.

Za razliko od časa vzpostavljanja povezave smo povečanje zakasnitve pri posredovanju paketov na stikalih napovedali (poglavje 4.6.2). Kot je že zapisano, je to povečanje posledica programske izvedbe stikal. V primeru strojne izvedbe stikal se ne pričakuje omembe vrednega povečanja zakasnitev.

Glede na omejitve uporabljenega simulacijskega okolja ocenjujemo, da zaradi nerealnih simulacijskih časov izvajanje obsežnejših simulacij, ki naj bi omogočile želeno aproksimacijo na večja realna omrežja, ni smiselno. Hkrati zaradi velike porabe virov izvajanje simulacij na večjih topologijah tudi ni izvedljivo.

Kot osrednji dokaz učinkovitosti metode z razpršenim reakcijskim krmiljenjem tako ostaja teoretični izračun karakteristik hipotetičnega omrežja v poglavju 4.4. Za pridobitev uporabnejših rezultatov bo potrebo zgraditi novo, ustreznejše simulacijsko okolje ali pa kar okolje, sestavljeno iz realnih komponent. Slednje tako ostaja izziv za bodoče delo.

7 SKLEP

Glavne značilnosti programsko opredeljenih omrežij so ločitev podatkovne in krmilne ravnine, enostavnejše naprave, centralizirano krmiljenje, avtomatizacija in virtualizacija ter prosti dostop oz. odprtost. Te značilnosti prinašajo vrsto prednosti v primerjavi s klasičnimi omrežji, kot na primer enostavnejše krmiljenje in upravljanje, večjo prožnost omrežja in možnost hitrejše vpeljave novih storitev, natančnejši nadzor in večjo varnost, neodvisnost in odprtost sistema ter nižjo ceno omrežja. Po drugi strani nova tehnologija še nima ustreznih odgovorov na pomanjkljivosti, ki izhajajo iz dosedanjih realizacij. Med njimi izstopajo slaba zmogljivost in razširljivost krmilnika ter pomanjkljivo izpolnjevanje zahtev po delovanju v realnem času, ki je posledica predolgega časa, potrebnega za vzpostavitev nove povezave.

Krmilnik omrežja SDN lahko izvaja krmiljenje oddaljenih stikal na dva oz. tri načine. Prvi je proaktivni način, pri katerem vnaprej vzpostavljene povezave zagotavljajo boljšo odzivnost omrežja, vendar ga zaradi omejene kapacitete stikal ni možno uporabljati za vse možne povezave. Drugi je reakcijski način, pri katerem sprotno vzpostavljanje povezav lahko traja predolgo. Tretji način je kombinirana uporaba prvega in drugega načina.

Poglobljena analiza dogajanja v primeru reakcijskega krmiljenja omrežja SDN je privedla do ideje, da bi krmiljenje lahko definirali nekoliko drugače in ga s tem optimizirali. Ideja temelji na ugotovitvi, da so si krmilna sporočila, ki preko omrežja vzpostavljajo isti podatkovni tok, med seboj zelo podobna. Nova metoda krmiljenja zato predlaga nov format krmilnega sporočila, ki s pomočjo ustrezne logike doseže množico stikal na poti novega podatkovnega toka. Pri tem nov podatkovni tok tudi vzpostavi. To novo metodo krmiljenja smo poimenovali razpršeno reakcijsko krmiljenje.

Glavna posledica krmiljenja množice stikal z enim samim krmilnim sporočilom je razbremenitev krmilnika pošiljanja velikega števila krmilnih sporočil. Tako prihranimo pri procesorskih virih na krmilniku in na razpoložljivi pasovni širini proti krmilniku. To pa pripomore k manjši obremenjenosti in boljši razširljivosti krmilne ravnine SDN. Na podatkovni ravnini z novo metodo dosežemo hitrejše vzpostavljanje tokovnih povezav, kar povečuje odzivnost omrežja SDN v realnem času.

Optimizacije v okviru opisanih karakteristik omrežja potrjujejo opravljeni teoretični izračuni, ki novo metodo primerjajo z osnovno in izboljšano osnovno metodo. Računsko smo pokazali, da nova metoda na omrežjih, ki bi imela povprečno dolžino tokovnih poti 20 stikal ter 5.000 novih tokovnih povezav vsako sekundo, razbremeni krmilnik kar za 82 odstotkov glede na izboljšano osnovno metodo. Čase vzpostavljanja novih podatkovnih povezav v takšnem omrežju pa nova metoda v povprečju zmanjša za 78 odstotkov.

S simulacijo tako velikega in tako dinamičnega omrežja zaradi omejitev okolja ni bilo mogoče preveriti. Tako smo s simulacijo pokazali, da nova metoda na omrežjih, ki bi imela povprečno dolžino tokovnih poti 3,5 stikal, razbremeni krmilnik za 7 odstotkov glede na izboljšano osnovno metodo. Izboljšanja časov vzpostavljanja novih podatkovnih povezav nam s simulacijo ni uspelo pokazati.

V bodoče zato velja postaviti novo okolje, ki bo omogočalo bolj realne pogoje in iz katerega bo možno razbrati rezultate, ki bodo bliže teoretično izračunanim. Ker se naše ugotovitve

glede razbremenitve krmilnika neposredno opirajo na zmanjšanje števila krmilnih sporočil, potrebnih za vzpostavljanje novih podatkovnih povezav, je bodoča naloga tudi zagotoviti možnost direktnega odčitka in primerjave obremenjenosti krmilnika. Poleg tega je v prihodnje potrebno predlagano metodo nadgraditi z mehanizmi, ki ji bodo zagotavljali tudi večjo varnost in robustnost.

8 PRILOGE

8.1 Programska koda Mininet

```
#!/usr/bin/python

# ##############################################################################
# ##
# ##  File:
# ##        eSDN_mininet.py
# ##
# ##  Project:
# ##        enhanced SDN: Optimizing of communication between controller and
# ##                  switches of SDN network by multicasting controller messages
# ##
# ##  Author:
# ##        Primoz Remic
# ##

from optparse import OptionParser

from mininet.node import RemoteController
from mininet.net import Mininet, CLI
from mininet.topo import LinearTopo
from mininet.log import setLogLevel
from netasm.back_ends.soft_switch.mininet.node import NetASMSwitch


def test_net():
    op = OptionParser()
    op.add_option('--cli', action="store_true", dest="cli")
    op.add_option('--switches', action="store", dest="switches")
    op.add_option('--hosts_per_switch', action="store", dest="hosts_per_switch")

    op.set_defaults(cli=False, switches=2, hosts_per_switch=1)
    options, args = op.parse_args()

    topo = LinearTopo(int(options.switches), int(options.hosts_per_switch))

    NetASMSwitch.CTL_ADDRESS = "127.0.0.1"
    NetASMSwitch.CTL_PORT = 7791

    net = Mininet(topo, switch=NetASMSwitch, autoSetMacs=True, autoStaticArp=True,
                  controller=lambda name: RemoteController(name))

    NetASMSwitch.start_datapath(net.switches, address="127.0.0.1", port=6633)
    net.start()

    if options.cli:
        CLI(net)
    else:
        net.pingAll()

    net.stop()
    NetASMSwitch.stop_datapath()


if __name__ == '__main__':
    # Tell mininet to print useful information
    setLogLevel('info')
    test_net()
```

8.2 Programska koda POX

```
# ##############################################################################
# ##
# ##  File:
# ##        eSDN_POX.py
# ##
# ##  Project:
# ##        enhanced SDN: Optimizing of communication between controller and
# ##                  switches of SDN network by multicasting controller messages
# ##
# ##  Author:
# ##        Primoz Remic
# ##

from pox.core import core
from pox.lib.util import dpidToStr
from pox.lib.packet.ethernet import ETHER_BROADCAST
from pox.openflow.libopenflow_01 import *
from ast import literal_eval
from optparse import OptionParser
from netasm.netasm.core.common import ports_to_bitmap
from netasm.back_ends.soft_switch.api import OutMessage, InMessage

import time

log = core.getLogger()
ETHADDR_MR = EthAddr(b"\x01\x80\xC2\x00\x00\xFF")   # 01-80-C2-00-00-FF
(unassigned)
ETHTYPE_MR = 0xFFEF                                 # Experimental

class InstalledRules(object):
    """
    Storage for installed rules in format {dest : {switch : output_port}}
    """
    def __init__(self):
        self.__rules = {}
    @property
    def rules(self):
        return self.__rules
    def add_rule(self, switch, dest, output_port):
        if dest in self.__rules:
            self.__rules[dest][switch] = output_port
        else:
            self.__rules[dest] = {switch:output_port}
    def get_rule(self, switch, dest):
        if dest in self.__rules:
            return self.__rules[dest].get(switch, 0)
        else:
            return 0
    def get_dests(self):
        return self.__rules.keys()
    def get_rule_reach(self, dest):
        return len(self.__rules[dest])
def _get_table_index(dst_host):
    # Calculate data plane table index
    return dst_host%32
def _getOutPort_linTopo(cur_sw, dest, k, n):
    """
    Calculate cur_sw's output port for link towards given destination in lin. topo.
    Args:
        cur_sw: current switch id
        dest: destination host id
        k: number of switches in topology
        n: number of hosts per switch
    Returns:
```

```
        corresponding output port id or 0 if not found
    """
    # Check constrains
    if dest < 1 or dest > k*n:
        return 0
    # Destination switch id:
    dest_sw = dest % k
    if not dest_sw:
        dest_sw = k
    if dest_sw < cur_sw:
        # towards lower switch
        return n+1
    elif dest_sw > cur_sw:
        # towards higher switch
        if cur_sw == 1:
            # first switch in a row
            return n+1
        else:
            # not first switch in a row
            return n+2
    else:
        # target host is on the current switch
        return (dest-dest_sw)/k + 1
def _getOutputPort(switch_id, host_id):
    # Calculate out port for linear topology.
    return _getOutPort_linTopo(switch_id, host_id, no_of_s, h_per_s)
def _getAT_linTopo(cur_sw, dest, dests_list, k, n):
    """
    Calculate best alternative trail towards given destination from list of
    available destinations for linear topology.
    Args:
        cur_sw: current switch id
        dest: required destination host id
        dests_list: list of destinations for which rules are already installed
        k: number of switches in topology
        n: number of hosts per switch
    Returns:
        alternative trail destination (host ID) to be used or 0 if not found
        + split point (the switch id where AT differs from original trail)
    """
    # Check constrains
    if dest < 1 or dest > k*n:
        #print "# destination out of topo"
        return 0
    # Destination switch id:
    dest_sw = dest % k
    if not dest_sw:
        dest_sw = k
    if dest_sw == cur_sw:
        #print "# target host is on the current switch - AT not needed"
        return 0
    if dest_sw < cur_sw:
        # towards lower switch
        direction = "down"
        next_sw = cur_sw - 1  # Next hop
    else:
        # towards higher switch
        direction = "up"
        next_sw = cur_sw + 1  # Next hop
    if next_sw == dest_sw:
        #print "# target host is on the next switch - AT not needed"
        return 0
    # Is there a rule for the dest on next switch?
    if dest in dests_list:
        r = i_rules.get_rule(next_sw, dest)
        if r:
```

```
            #print "# rule already installed on next switch - AT not needed"
            return 0
        else:
            dests_list.remove(dest)  # remove the dest from dests_list
    # Remove all destinations not installed on next switch from dests_list
    # -(do not promote what cannot be used on next hop)
    next_sw_dests = [i_rules.get_rule(next_sw, x) for x in dests_list]
    j = 0
    for i in range(0, len(next_sw_dests)):
        if next_sw_dests[i] == 0:
            dests_list.pop(i-j)
            j += 1              # evry pop shortens the list...
    if len(dests_list) == 0:
        #print "# no fitting trails..."
        return 0
    # Get most suitable destination to the AT
    AT_dest = 0 # Initialize Alternative Trail destination switch id
    AT_sp = 0   # Initialize Alternative Trail split point switch id
    dist = -k   # Initialize distance between the dest and AT_dest
    if direction == "up":
        for i in range(0, len(dests_list)):
            atc = dests_list[i]             # Alternative trail candidate
            atc_sw = atc%k if atc%k else k  # Alternative trail candidate switch id
            new_dist = atc_sw - dest_sw
            # Check if dest in list really goes 'up' and if it is better candidate
            if (atc_sw > cur_sw) and (new_dist > dist):
                dist = new_dist
                AT_dest = atc
                AT_sp = atc_sw
                if dist >= 0:
                    AT_sp = dest_sw
                    # stop the search - this one is good!
                    break
    else: # "down"
        for i in range(0, len(dests_list)):
            atc = dests_list[i]             # Alternative trail candidate
            atc_sw = atc%k if atc%k else k  # Alternative trail candidate switch id
            new_dist = dest_sw - atc_sw
            # Check if dest in list really goes 'down' and if it is better
candidate
            if (atc_sw < cur_sw) and (new_dist > dist):
                dist = new_dist
                AT_dest = atc
                AT_sp = atc_sw
                if dist >= 0:
                    AT_sp = dest_sw
                    # stop the search - this one is good!
                    break
    return [AT_dest, AT_sp]
def _getAlternativeTrail(crnt_switch_id, host_id):
    """
    Get alternative trail to be used as the path to the required host_id
    Args:
        crnt_switch_id:  current switch id (where we are)
        host_id:         required destination - host ID to get AT for
    Returns:
        alternative trail destination (host ID) to be used or 0 if not found
        + split point (the switch id where AT path differs from required path)
    """
    # Get all destinations stored in installed rules
    all_dests = i_rules.get_dests()
    if len(all_dests) == 0:
        #print "no dests available"
        return [0,0]
    # Calculate best alternative trail for linear topology.
    AT = _getAT_linTopo(crnt_switch_id, host_id, all_dests, no_of_s, h_per_s)
```

```
    if isinstance(AT, list):
        return AT
    else:
        return [0,0]
def _getNodes_linTopo(cur_sw, dest, k, n):
    """
    Compose a list of switches-ids along the path for linear topology.
    Args:
        cur_sw: current switch id
        dest: required destination host id
        k: number of switches in topology
        n: number of hosts per switch
    Returns:
        List of switches dpids from dest's switch towards crnt_switch_id
    """
    # Check constrains
    if dest < 1 or dest > k*n:
        #print "# destination out of topo"
        #return empty list
        return []
    # Destination switch id:
    dest_sw = dest % k
    if not dest_sw:
        dest_sw = k
    if dest_sw == cur_sw:
        #print "# target host is on the current switch"
        #return empty list
        return []
    if dest_sw < cur_sw:
        # towards lower switch (direction is down)
        return range(dest_sw, cur_sw)
    else:
        # towards higher switch (direction is up)
        return range(dest_sw, cur_sw, -1)
def _getNodesOnPath(crnt_switch_id, host_id):
    """
    Get list of all switches along path from crnt_switch_id towards host_id
    Args:
        crnt_switch_id:  current switch id (where we are)
        host_id:         required destination - host ID to get AT for
    Returns:
        List of switches dpid
    """
    # Calculate list of switches for linear topology.
    return _getNodes_linTopo(crnt_switch_id, host_id, no_of_s, h_per_s)
def _mcastRuleMsgPack(req_dst, req_dst_tbl_index, AT_dst, AT_dst_tbl_index):
    """
    Compose ethernet packet for multicast propagation of alternative trail.
    Packet format:
    ------------------------------------------------------------------------
    | eth_hdr (14o) | req_dst (6o) | req_idx (2o) | AT_dst (6o) | AT_idx (2o) |
    ------------------------------------------------------------------------
    Args:
        req_dst: required destination
        AT_dst:  alternative destination
        AT_dst_tbl_index: index of AT entry in match_table on switches
                          (has to be the same on all switches in order to be of use)
    Returns: Composed ethernet packet
    """
    def build_pyload():
        n=2
        # destinations are mac addresses - 12 digits long (+2x4 digits for indexes)
        data = "%012x%04x%012x%04x"%(req_dst, req_dst_tbl_index,
                                     AT_dst, AT_dst_tbl_index)
        s2c = [chr(int(data[i:i+n],16)) for i in range(0, len(data), n)]
        return ''.join(s2c)
```

```
    mr_e = ethernet(
        src=EthAddr("01:01:01:01:01:01"),
        dst=ETHADDR_MR,
        type=ETHTYPE_MR,
        payload=build_pyload()
    )
    return mr_e.pack()
def _handle_VendorIn(event):
    in_msg = InMessage(event.ofp)
    if in_msg.is_packet_in:
        if in_msg.reason == 'MATCH_TABLE_MISS':
            out_msg = OutMessage()
            rule_for_install = 1
            AT_dst = 0
            def install_rule(dpid, dst, port, con):
                if port <= 0: return
                if con is None: return
                # If rule not already installed...
                if i_rules.get_rule(dpid, dst) == 0:
                    # ...install it in the switch
                    tbl_index = _get_table_index(dst)
                    out_msg.add_table_entry('match_table', tbl_index,
                                            {'eth_dst': (dst, 0xFFFFFFFFFFFF)})
                    con.send(out_msg)
                    out_msg.add_table_entry('params_table', tbl_index,
                                            {'outport_bitmap': ports_to_bitmap(port)})
                    con.send(out_msg)
                    # Store installed rule
                    i_rules.add_rule(switch=dpid, dest=dst, output_port=port)
                else:
                    print "DENY rule: s%d : h%d -> port %d" % (dpid, dst, port)
            #get sending switch id
            switch_dpid = event.dpid
            print "switch=" + str(switch_dpid)
            #get incoming port
            in_port = in_msg.port
            #parse ethernet packet
            packet = event.parsed
            src_int = int(str(packet.src).replace(':', ''), 16)
            dst_int = int(str(packet.dst).replace(':', ''), 16)
            in_type = packet.getNameForType(packet.type)
            # Check if multicast rule was send back to controller
            if ('0x'+in_type) == str(hex(ETHTYPE_MR)):
                print "ERROR: MR packet received!"
                return
            # Resolve output port
            out_port = _getOutputPort(switch_dpid, dst_int)
            # Do not install rule for IPv6 or broadcast messages
            if  in_type == 'IPV6' or dst_int == 0xFFFFFFFFFFFF:
                rule_for_install = 0
            # Get data plane table index
            tbl_index = _get_table_index(dst_int)
            # If output port ressolved and in case of multicast rules usage...
            # Get alternative trail host id and path split point switch id
            if out_port and mcast_rules:
               AT_dst,AT_sp_dpid = _getAlternativeTrail(crnt_switch_id=switch_dpid,
                                                        host_id=dst_int)
            # Was AT found?
            if AT_dst:
                mr_msg = OutMessage()
                print "Installing MCAST_RULE..."
                # Get connection to AT split point
                AT_sp_con = core.openflow.getConnection(AT_sp_dpid)
                # Resolve AT_sp output port
                sp_out_port = _getOutputPort(AT_sp_dpid, dst_int)
                # Install rule in the AT SP switch
```

```
                # (switch where AT path differs from required path)
                install_rule(AT_sp_dpid, dst_int, sp_out_port, AT_sp_con)

                # Send multicast rule, to be installed in all switches along the AT
                mr_msg.data = _mcastRuleMsgPack(dst_int, tbl_index, AT_dst,
                                                _get_table_index(AT_dst))
                mr_msg.packet_out([out_port], mr_msg.data)
                event.connection.send(mr_msg)

            if (out_port and rules2all):
                # Get list of all switches on the path
                dpids = _getNodesOnPath(crnt_switch_id=switch_dpid,host_id=dst_int)
                for s_dpid in dpids:
                    # Get connection to s_dpid
                    s_con = core.openflow.getConnection(s_dpid)
                    # Resolve s_dpid output port
                    s_out_port = _getOutputPort(s_dpid, dst_int)
                    # Install rule in the s_dpid
                    install_rule(s_dpid, dst_int, s_out_port, s_con)

            if rule_for_install and out_port:
                # Force context switch...
                time.sleep(0)
                pass
             # Send packet to output port(s)
             if not out_port:
                # Broadcast if output port not resolved
                ports = []
                for i in range(1, PORT_COUNT + 1):
                    if i != in_msg.port:
                        ports.append(i)
                out_msg.packet_out(ports, in_msg.packet_data)
                event.connection.send(out_msg)
                return
             else:
                # Forward package to the right output port
                out_msg.packet_out([out_port], in_msg.packet_data)
                event.connection.send(out_msg)
             # Do not install rule for IPv6 or broadcast messages
             if  rule_for_install == 0:
                print "IGNORED!"
                return
             # Install a rule in the switch
             install_rule(switch_dpid, dst_int, out_port, event.connection)
        if in_msg.reason == 'MRPT_MISS':
            print "Error: MRPT_MISS!!!"

def _handle_ConnectionUp(event):
    msg = OutMessage()
    msg.set_policy("netasm.examples.remo.eSDN_netasm")
    event.connection.send(msg)
    # add match table entry for broadcast messages
    tbl_index = 0
    dst_int = 0xFFFFFFFFFFFF
    ports = []
    for i in range(1, PORT_COUNT + 1):
        ports.append(i)
    msg.add_table_entry('match_table', tbl_index,
                        {'eth_dst': (dst_int, 0xFFFFFFFFFFFF)})
    event.connection.send(msg)
    msg.add_table_entry('params_table', tbl_index,
                        {'outport_bitmap': ports_to_bitmap(ports)})
    event.connection.send(msg)
    log.info("netasm.examples.eSDN.eSDN_netasm for %s", dpidToStr(event.dpid))
def launch(switches=2, hosts_per_switch=1, method=1):
    # Store options values
```

```
    global rules2all
    global mcast_rules
    global no_of_s
    global h_per_s
    global PORT_COUNT
    no_of_s = int(switches)
    h_per_s = int(hosts_per_switch)
    PORT_COUNT = h_per_s + 2
    # Select method (default: method-I)
    rules2all = False
    mcast_rules = False

    if (method == str(2)):
        # method-II (send FLOW_MOD to all switches along the path)
        rules2all = True
    elif (method == str(3)):
        # method-III (use of MCAST_RULES)
        mcast_rules = True

    # Storage for installed rules
    global i_rules
    i_rules = InstalledRules()

    core.openflow.addListenerByName("ConnectionUp", _handle_ConnectionUp)
    core.openflow.addListenerByName("VendorIn", _handle_VendorIn)
    log.info("netasm.examples.eSDN.eSDN_netasm running.")
```

8.3 Programska koda NetASM

```
# ##############################################################################
# ##
# ##  File:
# ##        eSDN_netasm.py
# ##
# ##  Project:
# ##        enhanced SDN: Optimizing of communication between controller and
# ##                  switches of SDN network by multicasting controller messages
# ##
# ##  Author:
# ##        Primoz Remic
# ##

from netasm.netasm.core import *

def main():
    ### Declarations ###
    decls = Decls(TableDecls())
    ETHTYPE_MR = 0xFFEF
    ## Tables ##
    # Primary match table
    TABLE_SIZE = Size(32)
    decls.table_decls[TableId('match_table')] = \
        Table(TableFieldsCollection.MatchFields(),
            TABLE_SIZE,
            TableTypeCollection.CAM)
    match_table = decls.table_decls[TableId('match_table')]
    match_table.table_fields[Field('eth_dst')] = Size(48),
                                                 MatchTypeCollection.Binary
    # Primary params table
    decls.table_decls[TableId('params_table')] = \
        Table(TableFieldsCollection.SimpleFields(),
            TABLE_SIZE,
            TableTypeCollection.RAM)
    params_table = decls.table_decls[TableId('params_table')]
    params_table.table_fields[Field('outport_bitmap')] = Size(16)
```

```
# Note: outport_bitmap is a bitmap
# Mcast Rule match table
decls.table_decls[TableId('MR_match_table')] = \
    Table(TableFieldsCollection.MatchFields(),
        TABLE_SIZE,
        TableTypeCollection.CAM)
MR_match_table = decls.table_decls[TableId('MR_match_table')]
MR_match_table.table_fields[Field('req_dstination')] = Size(48),
                                                    MatchTypeCollection.Binary
# MR params table
decls.table_decls[TableId('MR_params_table')] = \
    Table(TableFieldsCollection.SimpleFields(),
        TABLE_SIZE,
        TableTypeCollection.RAM)
MR_params_table = decls.table_decls[TableId('MR_params_table')]
MR_params_table.table_fields[Field('AT_index')] = Size(16)

### Code ###
code = I.Code(
    ##################
    ### Arguments ####
    ##################
    Fields(),

    ##################
    ## Instructions ##
    ##################
    I.Instructions(

        ##################
        ## Parse packet ##
        ##################
        # Add ethernet header fields in the header set
        I.ADD(O.Field(Field('eth_dst')),
              Size(48)),
        I.ADD(O.Field(Field('eth_src')),
              Size(48)),
        I.ADD(O.Field(Field('eth_type')),
              Size(16)),
        # Add MR fields
        I.ADD(O.Field(Field('req_dst')), Size(48)),
        I.ADD(O.Field(Field('req_idx')), Size(16)),
        I.ADD(O.Field(Field('AT_idx')), Size(16)),
        I.ADD(O.Field(Field('index')), Size(16)),
        # Load fields with default values
        I.LD(O.Field(Field('eth_dst')),
             O.Value(Value(0, Size(48)))),
        I.LD(O.Field(Field('eth_src')),
             O.Value(Value(0, Size(48)))),
        I.LD(O.Field(Field('eth_type')),
             O.Value(Value(0, Size(16)))),
        I.LD(O.Field(Field('req_dst')), O.Value(Value(0, Size(48)))),
        I.LD(O.Field(Field('req_idx')), O.Value(Value(0, Size(16)))),
        I.LD(O.Field(Field('AT_idx')), O.Value(Value(0, Size(16)))),
        # Parse ethernet
        # load ethernet header fields from the packet
        I.LD(O.Field(Field('eth_dst')),
             O.Location(
                 Location(
                     O.Value(Value(0, Size(16))),
                     ))),
        I.LD(O.Field(Field('eth_src')),
             O.Location(
                 Location(
                     O.Value(Value(48, Size(16))),
                     ))),
```

```
I.LD(O.Field(Field('eth_type')),
     O.Location(
         Location(
             O.Value(Value(96, Size(16))),
             ))),
# Check if MR package
I.BR(O.Field(Field('eth_type')),
     Op.Neq,
     O.Value(Value(ETHTYPE_MR, Size(16))),
     Label('LBL_LKUP')),
# Parse MR fields from MR package
I.LD(O.Field(Field('req_dst')),
     O.Location(
         Location(
             O.Value(Value(112, Size(16))),
         ))),
I.LD(O.Field(Field('req_idx')),
     O.Location(
         Location(
             O.Value(Value(160, Size(8))),
         ))),
I.LD(O.Field(Field('AT_idx')),
     O.Location(
         Location(
             O.Value(Value(224, Size(8))),
         ))),

########################
## Handle MR packet   ##
########################
I.ATM(
    I.Code(
        Fields( Field('req_dst'),
                Field('req_idx'),
                Field('AT_idx'),
                Field('index'),
               ),
        I.Instructions(
            # Lookup and forward multicast rule (MR)
            # Lookup in the match table and drop MR if matched req_dst
            I.LKt(O.Field(Field('index')),
                  TableId('match_table'),
                  O.Operands_(
                      O.Field(Field('req_dst')))),
            I.BR(O.Field(Field('index')),
                 Op.Eq,
                 O.Value(Value(-1, Size(16))),
                 Label('LBL_MRFWD_1')),
            # Case: There is match for MR in match table (is AT SP)
            # do not forward the MR
            I.DRP(Reason('MR dropped - AT SP reached', '')),
            I.JMP(Label('LBL_HLT')),
            # Case: There is no match for MR in match table (not AT SP)
            I.LBL(Label('LBL_MRFWD_1')),
            # Store MR in the MR_match_table
            I.STt(TableId('MR_match_table'),
                  O.Field(Field('req_idx')),
                  O.OperandsMasks_(
                      (O.Field(Field('req_dst')),
                       Mask(0xFFFFFFFFFFFF)))),
            # Store AT index in the MR_params_table
            I.STt(TableId('MR_params_table'),
                  O.Field(Field('req_idx')),
                  O.Operands_(
                      O.Field(Field('AT_idx')))),
            # Forward the MR
```

```
            # Get AT output port, to forward MR to
            I.LDt(
                O.Operands__(
                    O.Field(Field('outport_bitmap'))),
                TableId('params_table'),
                O.Field(Field('AT_idx'))),

            ##########
            ## Halt ##
            ##########
            I.LBL(Label('LBL_HLT')),
            I.HLT()
        )
    )
),
I.JMP(Label('LBL_HLT')),

########################
## Lookup and forward ##
########################
I.LBL(Label('LBL_LKUP')),
I.ATM(
    I.Code(
        Fields(Field('eth_dst')),
        I.Instructions(
            # Add the following header fields in the header set
            I.ADD(O.Field(Field('index')),
                  Size(16)),
            I.ADD(O.Field(Field('index1')),
                  Size(16)),
            # Add temporary field for output port copy
            I.ADD(O.Field(Field('outport_bitmap_tmp')),
                  Size(16)),
            # Load fields with default values
            I.LD(O.Field(Field('outport_bitmap_tmp')),
                 O.Value(Value(0, Size(16)))),
            # Lookup in the match table and store the matched index
            I.LKt(O.Field(Field('index')),
                  TableId('match_table'),
                  O.Operands_(
                      O.Field(Field('eth_dst')))),
            I.BR(O.Field(Field('index')),
                 Op.Neq,
                 O.Value(Value(-1, Size(16))),
                 Label('LBL_PT_0')),
            # Case: there is no match in the match table
            # Lookup in the MR match table
            I.LKt(O.Field(Field('index1')),
                  TableId('MR_match_table'),
                  O.Operands_(
                      O.Field(Field('eth_dst')))),
            I.BR(O.Field(Field('index1')),
                 Op.Neq,
                 O.Value(Value(-1, Size(16))),
                 Label('LBL_ATFWD')),
            I.CTR(Reason('MATCH_TABLE_MISS', '')),
            I.JMP(Label('LBL_HLT')),
            # Case: there is a match in the MR match table
            # Forward packet via AT
            I.LBL(Label('LBL_ATFWD')),
            # Get AT index from the MR parameters table
            I.LDt(
                O.Operands__(
                    O.Field(Field('index'))),
                TableId('MR_params_table'),
                O.Field(Field('index1'))),
```

```
                    # Load output port from the l2 parameters table
                    I.LDt(
                        O.Operands__(
                            O.Field(Field('outport_bitmap'))),
                        TableId('params_table'),
                        O.Field(Field('index'))),
                    I.JMP(Label('LBL_HLT')),
                    # Case: there is a match in the match table
                    I.LBL(Label('LBL_PT_0')),
                    # Load output port and others from the l2 parameters table
                    I.LDt(
                        O.Operands__(
                            O.Field(Field('outport_bitmap_tmp'))),
                        TableId('params_table'),
                        O.Field(Field('index'))),
                    # Make sure that the packet is not send to incoming port
                    # (cause broadcast rule has all ones)
                    I.OP(
                        O.Field(Field('outport_bitmap')),
                        O.Field(Field('inport_bitmap')),
                        Op.And,
                        O.Field(Field('outport_bitmap_tmp')),
                    ),
                    I.OP(
                        O.Field(Field('outport_bitmap')),
                        O.Field(Field('outport_bitmap')),
                        Op.Xor,
                        O.Field(Field('outport_bitmap_tmp')),
                    ),

                    ##########
                    ## Halt ##
                    ##########
                    I.LBL(Label('LBL_HLT')),
                    I.HLT()
                )
            )
        ),
        ##########
        ## Halt ##
        ##########
        I.LBL(Label('LBL_HLT')),
        I.HLT()
    )
)
return Policy(decls, code)
```

Seznam uporabljenih virov

[1] B. N. Astuto, M. Mendonca, X. N. Nguyen, K. Obraczka, T. Turletti: "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *IEEE Communications Surveys & Tutorials*, št. 16, zv. 3, str. 1617–1634, 2014.

[2] T. Benson, A. Akella, D. Maltz, "Unraveling the complexity of network management," v zborniku *6th USENIX Symposiumon Networked Systems Design and Implementation*, ser. NSDI'09, str. 335–348, Berkeley, ZDA, 2009.

[3] P. Bosshart , D. Daly , G. Gibb , M. Izzard , N. McKeown , J. Rexford , C. Schlesinger, D. Talayco , A. Vahdat , G. Varghese , D. Walker, "P4: programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, št. 44, zv. 3, str. 87–95, New York, ZDA, 2014.

[4] P. Bosshart, G. Gibb, H. S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, M. Horowitz, "Forwarding metamorphosis: Fastprogrammable match-action processing in hardware for SDN," v zborniku *ACM SIGCOMM 2013 conference on SIGCOMM*, št. 43, zv. 4, str. 99–110, New York, ZDA, 2013.

[5] W. Braun, M. Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices," *Future Internet*, št. 6, zv. 2, str. 302–336, Basel, Švica, 2014.

[6] Y. Chiba, Y. Shinohara , H. Shimonishi: "Source flow: handling millions of flows on flow-based nodes," *ACM SIGCOMM Computer Communication Review*, št. 40, zv.4, str. 465–466, New York, ZDA, 2010.

[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, S. Banerjee, "DevoFlow: scaling flow management for high performance networks," v zborniku *ACM SIGCOMM 2011 conference*, str. 254–265, New York, ZDA, 2011.

[8] D. Erickson, "The Beacon OpenFlow controller," v zborniku *second ACM SIGCOMM workshop on Hot topics in software defined networking*, ser. HotSDN '13, str. 13–18, New York, ZDA, 2013.

[9] N. Feamster, J. Rexford, E. Zegura, "The Road to SDN: An Intellectual History of Programmable Networks," *ACMQueue – Large-Scale Implementations*, št. 11, zv.12, str. 87–98, New York, ZDA, 2013.

[10] M. P. Fernandez, "Comparing openflow controller paradigms scalability: Reactive and proactive," v zborniku *27th International Conference on Advanced Information Networking and Applications (AINA),* str. 1009–1016, Washington, ZDA, 2013

[11] P. Goransson, C. Black, *Software defined Networks: A Comprehensive Approach*. San Francisco: Morgan Kaufmann Publishers Inc, 2014.

[12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, št. 38, zv. 3, str. 105–110, New York, ZDA, 2008.

[13] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, P. Tran-Gia, "Modeling and performance evaluation of an OpenFlow architecture," v zborniku *23rd International Teletraffic Congress,* ser. ITC '11, str. 1–7, 2011.

[14] V. Jeyakumar, M. Alizadeh, Y Geng, C. Kim, D. Mazières, "Millions of little minions: using packets for low latency network programming and visibility," *ACM SIGCOMM Computer Communication Review*, št. 44, zv. 4, str. 3–14, Chicago, ZDA, 2014.

[15] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," v zborniku *IEEE*, št. 103, zv. 1, str. 14–76, 2014.

[16] D. Levin, A. Wundsam, B. Heller, N. Handigol, A. Feldmann, "Logically centralized? state distribution trade-offs in software defined networks," v zborniku *first workshop on Hot topics in software defined networks,* ser. HotSDN '12, str. 1–6, New York, ZDA, 2012.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Open-Flow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, št. 38, zv. 2, str. 69–74, New York, ZDA, 2008.

[18] D. Parniewicz, R. Doriguzzi Corin, L. Ogrodowczyk, M. Rashidi Fard, J. Matias, M. Gerola, V. Fuentes, U. Toseef, A. Zaalouk, B. Belter, E. Jacob, K. Pentikousis, "Design and implementation of an OpenFlow hardware abstraction layer," v zborniku *2014 ACM SIGCOMM Workshop on Distributed Cloud Computing,* ser. DCC'14, str. 71–76, New York, ZDA, 2014.

[19] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, S. Shenker, "Extending networking into the virtualization layer," v zborniku *workshop on Hot Topics in Networks,* ser. HotNets-VIII, 2009.

[20] V. D. Philip, Y. Gourhant, "Cross-control: A scalable multitopology fault restoration mechanism using logically centralized controllers," v zborniku *15th International Conference on High Performance Switching and Routing (HPSR),* str. 57–63, Vancouver, Kanada, 2014.

[21] K. Psounis, "Active networks: Applications, security, safety, and architectures," *IEEE Communications Surveys*, št. 2, zv. 1, str. 2–16, ZDA, 2009.

[22] C. Schlesinger, M. Greenberg, D. Walker, "Concurrent NetCore: from policies to pipelines," v zborniku *19th ACM SIGPLAN international conference on Functional programming,* str. 11–24, New York, ZDA, 2014.

[23] M. Shahbaz, N. Feamster, "The Case for an Intermediate Representation for Programmable Data Planes," v zborniku *1st ACM SIGCOMM Symposium on Software Defined Networking Research,* ser. SOSR '15, članek št. 3, New York, ZDA, 2015.

[24] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, G. Parulkar, "FlowVisor: A Network Virtualization Layer," *Deutsche Telekom Inc. R&D Lab, Stanford, Nicira Networks, Tech. Rep.*, ZDA, 2009.

[25] H. Song, "Protocol-oblivious Forwarding: Unleash the power of SDN through a future-proof forwarding plane," v zborniku *Second ACM SIGCOMM Workshop on Hot Topics in Software DefinedNetworking*, ser. HotSDN '13, str.127–132, New York, ZDA, 2013.

[26] B. Xiong, K. Yang, J. Zhao, W. Li, K. Li, "Performance evaluation of OpenFlow-based software-defined networks based on queueing model," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, zv. 102, št. C, str. 172–185, 2016.

Ostali viri

[27] A. Doria, J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, J. Halpern, "Forwarding and Control Element Separation (ForCES) Protocol Specification," *Internet Engineering Task Force*, RFC 5810, 2010. Dostopno na: http://www.ietf.org/rfc/rfc5810.txt (22. 8. 2016)

[28] N. Feamster, "Routing Control Platform," spletno izobraževanje *Coursera SDN 2015* modul 2.4. Dostopno na: https://www.youtube.com/watch?v=iFFSbBaKjg4, (22. 8. 2016)

[29] K. Greene, "TR10: Software-defined networking," *MIT Technology Review*, 2009. Dostopno na: http://www2.technologyreview.com/article/412194/tr10-software-defined-networking/, (26. 6. 2016)

[30] T. Huang, V. Jeyakumar, B. Lantz, B. O'Connor, N. Feamster, K. Winstein, A. Sivaraman, " Teaching Computer Networking with Mininet," *SIGCOMM 2014 Tutorial*. Dostopno na: http://conferences.sigcomm.org/sigcomm/2014/doc/slides/mininet-intro.pdf, (22. 8. 2016)

[31] H. Porat, "Reality Check: SDN – great in theory, less in practice". Dostopno na: http://www.rcrwireless.com/20160202/opinion/reality-check-sdn-great-in-theory-less-in-practice-tag10, (22. 8. 2016)

[32] Open Networking Funation, "Software-Defined Networking (SDN) Definition" . Dostopno na: https://www.opennetworking.org/sdn-resources/sdn-definition, (26. 6. 2016)

[33] Open Networking Funation, "OpenFlow Switch Specification Version 1.5.1". Dostopno na: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf, (22. 8. 2016)

[34] ONOS – Open Network Operating System (). Dostopno na: http://onosproject.org/, (22. 8. 2016)

[35] OpenDaylight, "OpenDaylight: A Linux Foundation Collaborative Project," Dostopno na: http://www.opendaylight.org, (26. 6. 2016)

[36] Ryu the Network Operating System. Dostopno na: https://ryu.readthedocs.io/en/latest/#, (22. 8. 2016)

[37] NOX. Dostopno na: http://www.noxrepo.org/, (22. 8. 2016)

[38] IEEE Standarts Association. Dostopno na: http://standards.ieee.org/develop/regauth/grpmac/public.html, (22. 8. 2016)

[39] IEEE 802 Numbers. Dostopno na: http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml, (22. 8. 2016)

[40] Floodlight, "Floodlight is an Open SDN controller". Dostopno na: http://www.projectfloodlight.org/floodlight/, (22. 8. 2016)

[41] POX Controller. Dostopno na: https://openflow.stanford.edu/display/ONL/POX+Wiki, (22. 2. 2016)

[42] Flowgrammable. Dostopno na: http://flowgrammable.org/sdn/openflow/, (22. 8. 2016)

[43] Open vSwitch. Dostopno na: http://openvswitch.org/, (22. 8. 2016)

[44] Ubuntu. Dostopno na: http://www.ubuntu.com/, (22. 8. 2016)

[45] VirtualBox. Dostopno na: https://www.virtualbox.org/, (22. 8. 2016)

[46] Vagrant. Dostopno na: https://www.vagrantup.com/, (22. 8. 2016)

[47] Mininet. Dostopno na: http://mininet.org/, (22. 8. 2016)

[48] Vagrant boxes. Dostopno na: http://www.vagrantbox.es/, (22. 8. 2016)

[49] MobaXterm. Dostopno na: http://mobaxterm.mobatek.net/, (22. 8. 2016)

[50] Coursera-sdn Vagrant box. Dostopno na: https://d396qusza40orc.cloudfront.net/sdn1/srcs/Vagrant%20Box/coursera-sdn-2015_64bit.box, (22. 8. 2016)

[51] Mininet download. Dostopno na: http://mininet.org/download/, (22. 8. 2016)

[52] B. Linkletter, "How to use MiniEdit, Mininet's graphical user interface, " *Blog*. Dostopno na: http://www.brianlinkletter.com/how-to-use-miniedit-mininets-graphical-user-interface/, (22. 8. 2016)

[53] Mininet Walkthrough. Dostopno na: http://mininet.org/walkthrough/, (22. 8. 2016)

[54] Mininet Walkthrough, "Start Wireshark". Dostopno na: http://mininet.org/walkthrough/#start-wireshark, (22. 8. 2016)

[55] Mininet Walkthrough, "Interact with Hosts and Switches". Dostopno na: http://mininet.org/walkthrough/#interact-with-hosts-and-switches, (22. 8. 2016)

[56] Click. Dostopno na: http://www.read.cs.ucla.edu/click/, (22. 8. 2016)

[57] NetASM-python. Dostopno na: https://github.com/NetASM/NetASM-python, (22. 8. 2016)